МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ»

А.Г. Смышляев, А.В. Жуков, И.В. Жуков

МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Лабораторный практикум
для студентов очной формы обучения направления
бакалавриата 09.03.02 – Информационные системы и технологии

Белгород 2016

УДК 004.056(07)
ББК 32.973-018.2я7
С50

Смышляев, А. Г.
С50 Методы и средства защиты информации: лабораторный практикум: учебное пособие / А.Г. Смышляев, А.В. Жуков, И.В. Жуков. – Белгород, 2016. – 102 с.

Учебное пособие предназначено для изучения студентами принципов построения блочных и потоковых шифров и особенностей применения криптографических пакетов и интерфейсов при реализации программных средств обеспечения информационной безопасности. Включает теоретический материал и задания к выполнению шести лабораторных работ.

Учебное пособие предназначено для студентов очной формы обучения направления бакалавриата 09.03.02 – Информационные системы и технологии.

Данное издание публикуется в авторской редакции.

УДК 004.056(07)
ББК 32.973-018.2я7

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Учебная дисциплина «Методы и средства защиты информации» изучается студентами направления бакалавриата 09.03.02 «Информационные системы и технологии» в четвертом семестре.

Задачей курса является формирование у студентов представления о принципах и существующих методиках в области защиты информационных систем от любых случайных или преднамеренных воздействий, представляющих угрозу их штатному функционированию, и, как следствие, создающих угрозы доступности, целостности и конфиденциальности используемых информационных ресурсов.

Одним из инструментов, используемых в мероприятиях по защите информационных ресурсов, является *криптография*. Данное пособие содержит теоретический материал и задания к выполнению шести лабораторных работ, в которых рассматриваются как вопросы самостоятельной реализации криптографических алгоритмов, так и особенности применения наиболее популярных криптографических пакетов и интерфейсов. Студентам предлагается выполнить программную реализацию потокового шифра на базе регистра сдвига с линейной обратной связью и блочного шифра ГОСТ 28147-89, являющего стандартом симметричного шифрования РФ. Кроме того, предлагается изучить особенности применения средств симметричного и асимметричного шифрования, реализованных в криптографических интерфейсах Microsoft CryptoAPI и Cryptography API: Next Generation и криптографическом пакете OpenSSL. Причем в пакете OpenSSL используются, в том числе, и отечественные алгоритмы шифрования, хэширования и электронной подписи. Также рассмотрены способы получения цифровых сертификатов X.509 и их применения при создании криптографических сообщений и реализации защищенной передачи данных по протоколу TLS.

Выполнение представленных в пособии лабораторных работ позволит студентам укрепить знание теоретического материала, а также получить практические навыки создания и применения средств криптографической защиты информации на базе собственных или библиотечных реализаций алгоритмов разных типов.

ПРАВИЛА ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ

По курсу «Информационная безопасность» предусмотрено выполнение ряда лабораторных работ. Большинство работ ориентированы на использование среды разработки для платформы Windows, поддерживающей создание программ на языке C/C++, такой как Visual Studio. Используемые в работах криптографические пакеты и интерфейсы либо являются частью операционной системы Windows, либо являются свободно распространяемым программным обеспечением, доступным в сети Интернет. Студент обязан перед выполнением каждой лабораторной работы самостоятельно ознакомиться с теоретическим материалом и по факту ее выполнения представить преподавателю результаты в виде работающего приложения и отчета. Все отчеты о выполнении лабораторных работ оформляются в печатном виде на листах формата А4. Отчет должен содержать:

1. Заголовок лабораторной работы – номер работы, данные о студенте, слова «Выполнение» и «Защита», название и цель работы.
2. Содержание работы.
3. Блок-схемы, представляющие алгоритм работы приложения в обобщенном виде.
4. Исходные коды приложений.
5. Результаты тестирования приложений.
6. Вывод о выполненной работе.

ЛАБОРАТОРНАЯ РАБОТА № 1

ПОТОКОВОЕ ШИФРОВАНИЕ ДАННЫХ

Цель работы: получить навыки в создании программной реализации алгоритма потокового шифрования на базе регистра сдвига с линейной обратной связью.

ОСНОВНЫЕ ПОНЯТИЯ

Потоковые шифры преобразуют открытый текст в шифртекст по одному биту за операцию. Простейшая реализация потокового шифра показана на рис. 1.1. **Генератор потока ключей** (иногда называемый генератором с бегущим ключом или генератор гаммы) выдает псевдослучайный поток битов: k_1, k_2, k_3, ..., k_i. Этот поток ключей (гамма шифра) и поток битов открытого текста, p_1, p_2, p_3, ..., p_i, подвергаются операции «исключающее или» (*XOR*), и в результате получается поток битов шифртекста

$$c_i = p_i \oplus k_i$$

При расшифровании операция XOR выполняется над битами шифртекста и тем же самым потоком ключей для восстановления битов открытого текста

$$p_i = c_i \oplus k_i$$

Так как

$$p_i \oplus k_i \oplus k_i = p_i,$$

данная схема работает правильно.

Безопасность системы полностью зависит от свойств генератора потока ключей. Если генератор потока ключей выдает бесконечную строку нулей, шифртекст будет совпадать с открытым текстом, и вся операция будет бессмысленна. Чем ближе выход генератора потока ключей к случайному, тем больше времени потребуется криптоаналитику, чтобы взломать шифр.

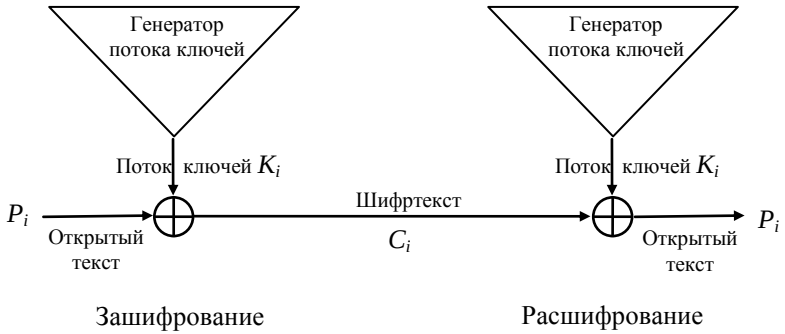

Рис. 1.1. Потоковый шифр

Большинство потоковых шифров основано на регистрах сдвига с обратной связью, которые состоят из двух частей: собственно сдвигового регистра и функции обратной связи (см. рис. 1.2).

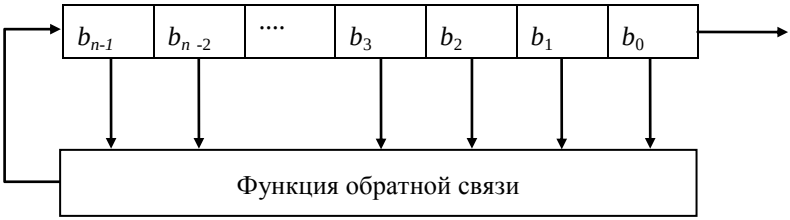

Рис. 1.2. Регистр сдвига с обратной связью

Сдвиговый регистр представляет собой последовательность битов. Количество битов определяется длиной регистра. Если длина равна *n* битам, то регистр называется *n*-битовым регистром сдвига. Всякий раз, когда нужно извлечь бит, все биты регистра сдвига сдвигаются вправо на 1 позицию. Новый крайний левый бит является функцией всех остальных битов регистра. На выходе регистра сдвига оказывается один, обычно младший значащий, бит. *Периодом* регистра сдвига называется длина получаемой последовательности до начала ее повторения.

К простейшему типу регистра сдвига с обратной связью относится регистр сдвига с линейной обратной связью (РСЛОС). Обратная связь представляет собой просто операцию *XOR* над некоторыми битами регистра. Перечень этих битов называется *отводной*

последовательностью. *n*-битовый РСЛОС может находиться в одном из 2^n-1 внутренних состояний (так как заполнение регистра нулями бесполезно).

Для представления отводной последовательности принято использовать двоичные многочлены вида $x^n + x^a + x^b + x^c + x^d + \ldots + x + 1$, где n – длина регистра, a, b, c, d и т.д. – номера битов, задающие последовательность отводов. Например:

$$x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$$

Кроме того, данный многочлен можно представить в виде (32, 7, 5, 3, 2, 1, 0). И эта, и предыдущая запись означает, что для данного 32-битового регистра сдвига новый бит генерируется с помощью операции *XOR* над седьмым, пятым, третьим, вторым, первым и нулевым битом (см. рис. 1.3).

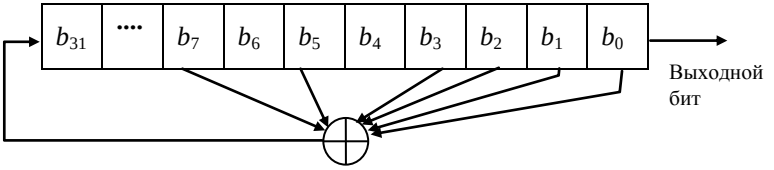

Рис. 1.3. 32-битовый РСЛОС

Для того чтобы конкретный РСЛОС имел максимальный период (т.е. циклически проходил через все 2^n-1 внутренних состояний), многочлен, образованный из отводной последовательности, должен быть примитивным (неприводимым) – т.е. не раскладываться на произведение двоичных многочленов меньшей степени. Некоторые неприводимые двоичные многочлены приведены в табл. 1.1.

Перед запуском РСЛОС при зашифровании или расшифровании, регистр необходимо инициализировать одним и тем же значением.

Таблица 1.1

Некоторые неприводимые двоичные многочлены

(1, 0)	(19, 5, 2, 1, 0)	(36, 11, 0)	(52, 3, 0)
(2, 1, 0)	(20, 3, 0)	(36, 6, 5, 4, 2, 1, 0)	(53, 6, 2, 1, 0)
(3, 1, 0)	(21, 2, 0)	(37, 6, 4, 1, 0)	(54, 8, 6, 3, 0)
(4, 1, 0)	(22, 1, 0)	(37, 5, 4, 3, 2, 1, 0)	(54, 6, 5, 4, 3, 2, 0)
(5, 2, 0)	(23, 5, 0)		(55, 24, 0)
(6, 1, 0)	(24, 4, 3, 1, 0)		(55, 6, 2, 1, 0)

Окончание табл. 1.1

(7, 1, 0) (7, 3, 0) (8, 4, 3, 2, 0) (9, 4, 0) (10, 3, 0) (11, 2, 0) (12, 6, 4, 1, 0) (13, 4, 3, 1, 0) (14, 5, 3, 1, 0) (16, 5, 3.2, 0) (17, 3, 0)	(25, 3, 0) (26, 6, 2, 1, 0) (27, 5, 2, 1, 0) (28, 3, 0) (29, 2, 0) (30, 6, 4, 1.0) (31, 3, 0) (31, 6, 0) (31, 7, 0) (32, 7, 6, 2, 0) (32, 7, 5, 3, 2, 1, 0)	(38, 6, 5, 1, 0) (39, 4, 0) (40, 5, 4, 3, 0) (41, 3, 0) (42, 7, 4, 3, 0) (42, 5, 4, 3, 2, 1, 0) (43, 6, 4, 3, 0) (44, 6, 5, 2, 0) (45, 4, 3, 1, 0) (46, 8, 7, 6, 0) (48, 9, 7, 4, 0) (48, 7, 5, 4, 2, 1, 0)	(56, 7, 4, 2, 0) (57, 7, 0) (57, 5, 3, 2, 0) (58, 19.0) (58, 6, 5, 1, 0) (59, 7, 4, 2, 0) (59, 6, 5, 4, 3, 1, 0) (60, 1, 0) (61, 5, 2, 1, 0) (63, 1, 0) (64, 4, 3, 1, 0)

СОДЕРЖАНИЕ РАБОТЫ

Разработать на выбранном языке программирования консольное или оконное приложение, реализующее описанный выше алгоритм для шифрования содержимого текстового или двоичного файла. Программа должна запрашивать имя входного и выходного файлов, представление образующего многочлена и инициализирующее значение. Разрядность РСЛОС должна быть меньше или равной максимальной разрядности стандартных целочисленных типов данных (64 бит).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Принципы работы потоковых шифров.
2. Какие операции используются при реализации потоковых шифров?
3. Что такое гамма шифра?
4. Что представляет собой регистр сдвига с линейной обратной связью?
5. Что такое отводная последовательность, и в какой форме ее можно представить?
6. Что такое период регистра сдвига?
7. Какие условия должны соблюдаться для того, чтобы РСЛОС имел максимальный период?

ЛАБОРАТОРНАЯ РАБОТА № 2

АЛГОРИТМ БЛОЧНОГО ШИФРОВАНИЯ ДАННЫХ ГОСТ 28147-89

Цель работы: научиться реализовывать на выбранном языке программирования алгоритм блочного шифрования данных ГОСТ 28147-89.

ОСНОВНЫЕ ПОНЯТИЯ

Криптоалгоритм ГОСТ 28147-89 является стандартом блочного шифрования РФ. Алгоритм имеет следующие параметры:

– *размер блока* – 64 бита;

– *размер ключа* – 256 бит. Ключ представляется как массив из восьми 32-битных подключей $K=\{K_0,K_1,\ldots,K_7\}$;

– *количество раундов* – 32 в режиме шифрования и 16 в режиме выработки имитовставки;

– *количество S-блоков* – 8. Каждый *S*-блок (в терминах стандарта – узел замены) содержит 16 четырехбитных значений, представляющих собой произвольную перестановку чисел от 0 до 15. Совокупность всех *S*-блоков можно представить в виде матрицы размером (8×16), называемой в терминах стандарта таблицей замен.

Алгоритм является вариантом сети Фейстеля, что предполагает одинаковую структуру каждого раунда (в терминах алгоритма – основного шага). Входом очередного раунда является выход предыдущего. Функция раунда, которая применяется к правой половине входного блока и всех промежуточных результатов состоит из следующих этапов:

– *сложение правой половины блока с подключом раунда по модулю* 2^{32}. При зашифровании подключи из массива *K* используются в следующем порядке: $K_0,K_1,\ldots,K_7,\ K_0,\ldots,K_7,K_0,\ldots,K_7,K_7,K_6,\ldots,K_0$, при расшифровании – в обратном порядке;

– *подстановка через S-блок*. 32-битовый результат сложения с подключом раунда представляется как 8 четырехбитовых фрагментов, каждый из которых заменяется тем элементом соответствующего ему *S*-блока, номер которого равен значению данного фрагмента;

– *циклический сдвиг результата подстановки на 11 разрядов влево*.

Результат вышеизложенных преобразований складывается по модулю 2 с левой половиной выхода предыдущего раунда. Сумма становится правой половиной выхода текущего раунда. Левой половиной становится правая половина выхода предыдущего раунда.

Для зашифрования 64-битового блока открытого текста выполняются 32 раунда. При этом в соответствии со структурой сети Фейстеля, перестановка левой и правой половин блока после последнего раунда не производится. Расшифрование производится по той же схеме, что и зашифрование, за исключением того, что подключи используются в обратном порядке.

Стандарт определяет три режима шифрования и один режим выработки имитовставки. К режимам шифрования относятся:

– *простая замена*. Данный режим предполагает независимое зашифрование/расшифрование 64-битных блоков открытого текста/шифртекста с помощью рассмотренного выше алгоритма. Поскольку алгоритм обрабатывает только 64-битные блоки, то в случае зашифрования неполного последнего блока необходимо организовать его дополнение. Сам стандарт не задает способ дополнения. Существуют несколько различных подходов к выполнению данной операции. Например, в качестве дополнения может использоваться случайная последовательность нулей и единиц. Последний байт дополнения должен содержать его размер, т.е. количество байт, которые необходимо отбросить при расшифровании. В том случае, если изначально размер открытого текста кратен 8 байтам, производится дополнение целым блоком. Еще одним вариантом является способ, предложенный в стандарте PKCS#7, разработанном компанией RSA Security, Inc. Согласно ему все дополняемые байты (вплоть до полного блока) заполняются одним и тем же числом, представляющим собой размер этого дополнения в байтах. Естественно, они также должны быть отброшены при расшифровании;

– *гаммирование*. Этот режим позволяет шифровать данные размером менее 64 бит. Базовый алгоритм используется в составе рекуррентного генератора псевдослучайных чисел. Вырабатываемая им гамма накладывается на блок открытого текста (или шифртекста при расшифровании) с помощью операции поразрядного «исключающего или» так же, как и в потоковых шифрах. Генератор инициализируется с помощью произвольного начального 64-битового значения, называемого *вектором инициализации* или *синхропосылкой*. В начале синхропосылка зашифровывается с помощью базового алгоритма. Далее результат этого зашифрования разбивается на

старшую и младшую 32-битные части. Младшая часть складывается с константой 1010101_{16} по модулю 2^{32}, а старшая с константой 1010103_{16} по модулю $(2^{32} – 1)$ с дальнейшим прибавлением единицы. Результаты сложения вновь поступают на вход генератора и процесс повторяется. Одновременно эти две 32-битные части объединяются в 64-битный блок, который зашифровывается с помощью базового алгоритма. Полученное таким образом значение используется в качестве гаммы;

– *гаммирование с обратной связью*. Данный режим, так же как и предыдущий, предполагает наложение гаммы на блок открытого текста или шифртекста с помощью операции «исключающее или». Однако вырабатываемая здесь гамма зависит не только от ключа, но и от всего открытого текста. Режим также предполагает наличие синхропосылки. В качестве первого элемента гаммы используется зашифрованная базовым алгоритмом синхропосылка. Он складывается по модулю два с первым блоком открытого текста, в результате чего получается первый блок шифртекста. Для зашифрования второго блока открытого текста используется элемент гаммы, полученный в результате применения базового алгоритма к первому блоку шифртекста и т.д. Таким образом, на шифрование текущего блока влияют результаты шифрования всех предыдущих. Расшифрование производится аналогичным образом. В данном режиме, как и в предыдущем, базовый алгоритм всегда работает в режиме зашифрования.

СОДЕРЖАНИЕ РАБОТЫ

Разработать на выбранном языке программирования консольное или оконное приложение, реализующее алгоритм ГОСТ 28147-89 в следующих режимах шифрования:

- режим простой замены (с дополнением блоков);
- режим гаммирования;
- режим гаммирования с обратной связью.

Программа должна запрашивать имя входного и выходного файлов, ключ, вектор инициализации (синхропосылку), режим работы (зашифрование или расшифрование), режим шифрования.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое сеть Фейстеля? Каковы основные принципы работы блочных шифров, устроенных по принципу сети Фейстеля?

2. Назовите все режимы шифрования, определенные в ГОСТ 28147-89.
3. Каковы разрядности блока и ключа в алгоритме ГОСТ 28147-89?
4. Что представляют собой таблицы замен (S-блоки) в алгоритме ГОСТ 28147-89?
5. Что представляет собой один раунд (основной шаг) алгоритма ГОСТ 28147-89?
6. Как может производиться дополнение неполных блоков в режиме простой замены?
7. Каковы недостатки режима простой замены?
8. Что собой представляет режим гаммирования?
9. Что собой представляет режим гаммирования с обратной связью?
10. Как функционирует схема зашифрования алгоритма ГОСТ 28147-89?
11. Как функционирует схема расшифрования алгоритма ГОСТ 28147-89?
12. Что такое синхропосылка?

ЛАБОРАТОРНАЯ РАБОТА № 3

СИММЕТРИЧНОЕ ШИФРОВАНИЕ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ КРИПТОГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ MICROSOFT CRYPTOAPI И CRYPTOGRAPHY API: NEXT GENERATION

Цель работы: ознакомиться с понятием криптопровайдера в интерфейсе CryptoAPI и провайдера алгоритма в интерфейсе Cryptography API: Next Generation, способами подключения к ним, получить навыки выполнения базовых криптографических преобразований: хэширования и симметричного шифрования.

ОСНОВНЫЕ ПОНЯТИЯ

Интерфейс CryptoAPI

В операционных системах (ОС) компании Microsoft, начиная с Windows 95, обеспечивается реализация шифрования, генерации ключей, создания и проверки цифровых подписей и других криптографических преобразований. Эти функции необходимы для работы операционной системы, однако ими может воспользоваться и любая прикладная программа. Для этого используется интерфейс прикладного программирования *CryptoAPI*.

Все современные операционные системы Windows поддерживают криптографический интерфейс CryptoAPI 2.0. Он содержит функции, осуществляющие базовые криптографические преобразования, а также дополнительные средства, такие как функции для работы с сертификатами X.509. Набор функций для осуществления базовых криптографических преобразований также называют CryptoAPI 1.0. Ниже будут рассмотрены некоторые функции именно этого интерфейса.

Все функции интерфейса CryptoAPI 1.0 содержатся в библиотеке *advapi32.dll*. Однако они выполняют лишь ряд вспомогательных операций и вызывают библиотеку, в которой непосредственно реализованы соответствующие криптографические преобразования. Такие библиотеки называются *криптопровайдерами* (*Cryptographic Service Providers, CSP*). Криптопровайдеры имеют стандартный набор из 23 обязательных и 2 необязательных функций. Программная часть криптопровайдера представляет собой *dll*-файл, подписанный

Microsoft; периодически Windows проверяет цифровую подпись, что исключает возможность подмены криптопровайдера. В составе Windows могут быть установлены криптопровайдеры разработанные не только Microsoft, но и сторонними производителями. Сведения обо всех установленных криптопровайдерах содержатся в системном реестре (*HKLM\SOFTWARE\Microsoft\Cryptography\Defaults\Provider*).

Каждый криптопровайдер поддерживает свою ключевую базу, которая представляется набором ключевых контейнеров. Каждый контейнер имеет имя, которое присваивается ему при создании, а затем используется для работы с ним. В ключевом контейнере сохраняется долговременная ключевая информация, например пары закрытый/открытый ключ для создания и проверки электронной подписи (ЭП). Сеансовые ключи для блочного или потокового шифрования в контейнере не хранятся.

Каждый криптопровайдер характеризуется собственным именем и типом. Имя это строка символов, по которой система распознает криптопровайдера. Например, базовый криптопровайдер, имеющийся в составе ОС Windows любой версии называется «Microsoft Base Cryptographic Provider v1.0». Тип криптопровайдера это целое число (тип *DWORD*), значение которого идентифицирует набор поддерживаемых алгоритмов цифровой подписи и обмена сеансовыми ключами. Криптопровайдер «Microsoft Base Cryptographic Provider v1.0» имеет тип 1, что соответствует символической константе *PROV_RSA_FULL*. В криптопровайдерах этого типа используется алгоритм RSA и для создания электронной подписи и для обмена сеансовыми ключами. В системе могут быть установлены криптопровайдеры разных типов. Также может быть установлено несколько криптопровайдеров одного типа.

В системном реестре содержится информация не только об именах криптопровайдеров и их типах, но и о том какой криптопровайдер использовать, если пользователь при вызове соответствующей функции не определил конкретное имя, а указал только требуемый ему тип. Такие криптопровайдеры называются *используемыми по умолчанию*. Например, для типа 1, в ОС, предшествующих Windows 2000, криптопровайдером по умолчанию являлся «Microsoft Base Cryptographic Provider v1.0». Начиная с Windows 2000, в состав ОС включены еще 2 криптопровайдера того же типа: «Microsoft Enhanced Cryptographic Provider» и «Microsoft Strong Cryptographic Provider», один из которых назначается криптопровайдером типа 1 по умолчанию (обычно «Microsoft Strong Cryptographic Provider»).

Криптопровайдеры Microsoft 1-го типа поддерживают уже устаревшие на данный момент алгоритмы шифрования RC2, RC4, DES, хэширования MD5, SHA-1. Современные алгоритмы: шифрования AES (с длинами ключа 128, 192, 256 бит) и хэширования SHA-2 (с длинами хэш-кода 256, 384, 512 бит) поддерживаются только криптопровайдерами типа 24 (*PROV_RSA_AES*). Начиная с версии Windows XP SP3, в состав операционных систем Microsoft включается единственный криптопровайдер этого типа «Microsoft Enhanced RSA and AES Cryptographic Provider». Он, также как и криптопровайдеры типа 1, использует алгоритм RSA для обмена ключами и создания ЭП. Расположены при этом все вышеперечисленные криптопровайдеры в составе одного файла – *rsaenh.dll*.

Для использования функции CryptoAPI в прикладной программе, разработчик должен объявить ее в программе как внешнюю, с указанием файла, в котором она находится. Чтобы облегчить процесс разработки, объявления необходимых типов, символических констант (таких как PROV_RSA_AES) и прототипов функций объединили в заголовочном файле *wincrypt.h*, который вошел в состав пакета Platform (Windows) SDK. По умолчанию пакет устанавливается в составе таких сред программирования, как C++ Builder и Visual Studio. Также он доступен для бесплатной загрузки на сайте компании Microsoft. Для использования в программе функций CryptoAPI необходимо включить в ее текст директивой *#include* содержимое файла *wincrypt.h*.

Рассмотрим функции CryptoAPI, необходимые для выполнения задания. Все описания и примеры будут приведены для языка программирования C/C++, применительно к консольному приложению (в оконных приложениях отличия только в способе вывода диагностических сообщений). Применяемые типы данных определены в файлах *wincrypt.h* и *windef.h*. Последний традиционно включается в текст в числе многих файлов, необходимых для разработки приложения Win32 API, при указании директивы *#include <windows.h>*.

В рассматриваемых далее примерах будут использоваться переменные, описание которых приведено ниже:

```
HCRYPTPROV hProv; //дескриптор криптопровайдера
HCRYPTKEY hKeyH; //дескриптор ключа, созданного из хэш-кода
HCRYPTHASH hHash; //дескриптор хэш-объекта
BYTE aBuf[512]; //буфер для данных (размер произвольный, кратный 16)
DWORD dwBuflen; //длина буфера
TCHAR szPass[100]; //строка для пароля
```

Назначение переменных будет пояснено по мере рассмотрения тех функций CryptoAPI, которые понадобятся при выполнении данной лабораторной работы.

Полное описание этих и других функций можно найти в разделе MSDN, посвященном использованию CryptoAPI (https://msdn.microsoft.com/en-us/library/windows/desktop/aa380256%28v=vs.85%29.aspx).

CryptAcquireContext

Эта функция выполняет подключение к криптопровайдеру. Ее прототип имеет вид:

```
BOOL WINAPI CryptAcquireContext(HCRYPTPROV *phProv,
                                LPCTSTR pszContainer,
                                LPCTSTR pszProvider,
                                DWORD dwProvType,
                                DWORD dwFlags);
```

В Win32 API (в том числе и в CryptoAPI) традиционно библиотечные функции, имеющие в качестве параметров строки, представлены в двух вариантах: с однобайтными ANSI-строками и «широкими» строками UNICODE. Здесь показан обобщенный прототип функции (макрос), который после обработки программы препроцессором будет заменен, в соответствии с настройками проекта, на нужный вариант (с буквой «A» или «W» в конце).

Функция возвращает дескриптор криптопровайдера в параметре *phProv*. В дальнейшем этот параметр будет передаваться другим функциям.

Параметр *pszContainer* содержит имя ключевого контейнера. Если приложение не будет использовать ключевой контейнер, то параметр задается как *NULL*. В частности, этот параметр должен быть нулевым, если приложение не будет производить экспорт/импорт сеансовых ключей. В противном случае имя контейнера необходимо задавать.

Параметр *pszProvider* содержит имя того криптопровайдера, к которому производится подключение. Параметр *dwProvType* определяет тип криптопровайдера. При вызове функции вместо него можно указывать символические константы, например, *PROV_RSA_AES*. Если в качестве параметра *pszProvider* указать *NULL*, то будет осуществлено подключение к криптопровайдеру, используемому по умолчанию для типа, указанного в параметре *dwProvType*.

Параметр *dwFlags* содержит значения флага (здесь и при описании некоторых других функций будут приведены только некоторые

значения флагов; полное описание функций можно найти в соответствующей литературе или библиотеке MSDN). Если задать его равным константе CRYPT_VERIFYCONTEXT, то получение контекста произойдет без подключения к контейнеру ключей. Для создания нового контейнера ключей с именем, заданным в параметре *pszContainer* используется флаг CRYPT_NEWKEYSET. Нулевое значение флага используется для подключению к уже существующему контейнеру.

Все функции CryptoAPI имеют логический тип (*BOOL*) и возвращают *TRUE* в случае успешного завершения и *FALSE* в противном случае. Поэтому при каждом вызове необходимо проверять возвращаемое функцией значение. В случае ошибки ее код можно узнать с помощью функции ***GetLastError***. Один из возможных вариантов обработки ошибки выполнения функции, получающей контекст без подключений к ключевому контейнеру, приведен ниже.

```
if (!CryptAcquireContext(&hProv, NULL, NULL,
    PROV_RSA_AES, CRYPT_VERIFYCONTEXT))
    {
       _tprintf(TEXT("Ошибка подключения к\
                криптопровайдеру. Код ошибки:0x%X"),
                GetLastError());
    }
```

Данный вариант вызова функции ***CryptAcquireContext*** выполнит подключение к криптопровайдеру типа 24 по умолчанию (то есть «Microsoft Enhanced RSA and AES Cryptographic Provider»).

Другой вариант вызова функции производит подключение к контейнеру ключей «my_cont», а в случае отсутствия создает его заново:

```
TCHAR szBuf[100];
if (!CryptAcquireContext(&hProv,TEXT("my_cont"),
                         NULL, PROV_RSA_AES, 0))
{
    DWORD err=GetLastError();
    if(err!=NTE_BAD_KEYSET)// Прочая ошибка
    {
         _tprintf(TEXT("Ошибка подключения к\
                  криптопровайдеру. Код ошибки:0x%X"),
                  err);
    }
    else // Создаем контейнер ключей «my_cont»
```

```
    if (!CryptAcquireContext(&hProv, TEXT("my_cont"),
                             NULL, PROV_RSA_AES,
                             CRYPT_NEWKEYSET))
    {
         _tprintf(TEXT("Ошибка создания контейнера ключей.\
                  Код ошибки:0x%X"), GetLastError());
    }
}
```

После того, как произведено подключение к нужному криптопровайдеру, для выполнения задачи симметричного шифрования необходимо получить сессионный ключ. Он может быть импортирован или сгенерирован. В свою очередь генерация ключа может быть произведена двумя способами: с помощью генератора псевдослучайных чисел или на основе хэшированной парольной фразы. Рассмотрим более подробно второй способ.

Однонаправленные хэш-функции выполняют свертку входных данных произвольной длины в битовую последовательность фиксированной длины, называемую хэш-кодом или дайджестом. При этом определение сообщения (прообраза), которое приводит к созданию данного хэш-кода, является трудновыполнимой (с точки зрения необходимых вычислительных ресурсов) задачей. А изменение всего лишь одного бита исходного сообщения приводит к кардинальным изменениям в его хэш-коде. Таким образом, пользователь, придумав парольную фразу и получив ее дайджест, использует его в качестве ключа симметричного шифрования.

В криптопровайдре «Microsoft Enhanced RSA and AES Cryptographic Provider» реализована поддержка алгоритма хэширования SHA-2 с переменной длиной дайджеста: 256, 384, или 512 бит. Для создания ключа с помощью хэш-кода используется функция ***CryptDeriveKey***. Однако перед ее вызовом необходимо, чтобы в памяти был создан хэш-объект, содержащий какой-либо хэш-код. Поэтому вначале рассмотрим функции для работы с хэш-объектом.

CryptCreateHash

Эта функция создает в памяти хэш-объект и имеет следующий прототип:

```
BOOL WINAPI CryptCreateHash(HCRYPTPROV hProv,
                            ALG_ID Algid,
                            HCRYPTKEY hKey,
                            DWORD dwFlags,
                            HCRYPTHASH *phHash);
```

Через параметр *hProv* в функцию передается дескриптор криптопровайдера. Параметр *Algid* представляет собой идентификатор используемого алгоритма хэширования. При использовании криптопровайдера типа 24 целесообразно использовать один из следующих алгоритмов: SHA-256, SHA-384, SHA-512. Соответственно определены следующие константы: *CALG_SHA_256*, *CALG_SHA_384*, *CALG_SHA_512*. Именно их можно указывать при вызове функции в качестве второго параметра. Однако их описание может отсутствовать в файле *wincrypt.h*, если он входит в состав Platform SDK устаревшей версии. В этом случае константы можно описать вручную:

```
#define CALG_SHA_256 0x800C
#define CALG_SHA_384 0x800D
#define CALG_SHA_512 0x800E
```

Параметр *hKey* содержит дескриптор ключа и используется в том случае, если должен быть создан код аутентичности сообщения (HMAC). В нашем случае этот параметр должен быть равен нулю. Параметр *dwFlags* содержит флаги и должен быть равен нулю. Функция копирует дескриптор созданного хэш-объекта по адресу, переданному через параметр *phHash*.

Приведем пример вызова функции. В этом примере и во всех последующих для краткости будут приведены только сами вызовы функций без проверки успешности их завершения. Однако в реальном приложении эти проверки необходимо осуществлять всегда, аналогично варианту, рассмотренному для функции ***CryptAcquireContext***.

```
CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash);
```

Функция ***CryptCreateHash*** создает пустой хэш-объект. Подать данные на вход хэш-объекта и собственно вычислить хэш-код позволяет функция ***CryptHashData***.

CryptHashData

Функция позволяет добавлять данные к объекту хэш-функции. Она может вызываться несколько раз, если данные, от которых вычисляется хэш, разбиты на блоки. Прототип функции имеет вид:

```
BOOL WINAPI CryptHashData(HCRYPTHASH hHash,
                          BYTE *pbData,
                          DWORD dwDataLen,
                          DWORD dwFlags);
```

В качестве параметра *hHash* передается дескриптор хэш-объекта, созданный функцией ***CryptCreateHash***. Параметр *pbData* содержит указатель на хэшируемые данные, а *dwDataLen* содержит размер этих данных в байтах. Параметр *dwFlags* должен быть равен нулю.

Приведем пример вызова функции, осуществляющего хэширование некоторого пароля, хранящегося в строке *szPass* (описание см. выше).

```
.....................................................
//Получение каким-либо образом пароля и занесение его в szPass.
.....................................................
CryptHashData(hHash, (PBYTE)szPass,
              _tcslen(szPass)*sizeof(TCHAR), 0);
```

После того, как произведено хэширование парольной фразы, необходимо вызвать функцию ***CryptDeriveKey*** (описание см. ниже). После формирования ключа хэш-объект должен быть уничтожен функцией ***CryptDestroyHash***.

CryptDestroyHash

Функция уничтожает хэш-объект, ранее созданный функцией ***CryptCreateHash***. Ее прототип имеет вид:

```
BOOL WINAPI CryptDestroyHash(HCRYPTHASH hHash);
```

Единственный параметр представляет собой дескриптор уничтожаемого объекта. Пример вызова:

```
CryptDestroyHash(hHash);
```

CryptDeriveKey

Функция создает сеансовый ключ на базе хэшированного пароля. Прототип функции имеет вид:

```
BOOL WINAPI CryptDeriveKey(HCRYPTPROV hProv,
                           ALG_ID Algid,
                           HCRYPTHASH hBaseData,
                           DWORD dwFlags,
                           HCRYPTKEY *phKey);
```

Параметр *hProv* содержит дескриптор криптопровайдера, полученный с помощью функции ***CryptAcquireContext***. Параметр *Algid* содержит идентификатор того алгоритма шифрования, для которого создается ключ. При использовании криптопровайдера типа 24 целесообразно использовать следующие варианты: AES-128, AES-192, AES-256. Соответственно определены константы: *CALG_AES_128*, *CALG_AES_192*, *CALG_AES_256*.

Параметр *hBaseData* содержит дескриптор хэш-объекта, содержащего хэшированный пароль. Причем после создания сеансового ключа в хэш-объект запрещено добавлять новые данные Параметр *dwFlags*, содержащий значения флагов, можно установить равным нулю. Функция копирует дескриптор созданного ключа по адресу, переданному через параметр *phKey*. Пример вызова:

```
CryptDeriveKey(hProv, CALG_AES_128, hHash, 0, &hKeyH);
```

Данный вызов создаст сессионный ключ, представляющий собой объект в защищенной области памяти, доступ к которому осуществляется через дескриптор. При этом объект содержит в себе не только сам ключ, но и параметры шифрования: алгоритм, режим (по умолчанию CBC), вектор инициализации или синхропосылку (по умолчанию нулевой), размер блока и т.д. Непосредственно в открытом виде ключ приложению недоступен.

После того, как создан сеансовый ключ, можно осуществлять зашифрование или расшифрование с помощью функций ***CryptEncrypt*** или ***CryptDecrypt*** (описание см. ниже), передавая им в качестве одного из параметров полученный дескриптор. После шифрования ключ необходимо уничтожить функцией ***CryptDestroyKey***. В дальнейшем такой ключ может быть воссоздан, если предоставить для хэширования первоначальную парольную фразу.

CryptDestroyKey

Функция освобождает ранее определенный дескриптор ключа, созданного любым из двух способов или дескриптор ключевой пары. Ее прототип имеет вид:

```
BOOL WINAPI CryptDestroyKey(HCRYPTKEY hKey);
```

Если параметром функции является дескриптор сессионного ключа, то при вызове уничтожается сам ключ. Если дескриптор ключевой пары, то происходит только его освобождение, а сама пара остается в контейнере.

Пример вызова функции:

```
CryptDestroyKey(hKeyH);
```

CryptEncrypt

Функция осуществляет зашифрование данных. Прототип имеет вид:

```
BOOL WINAPI CryptEncrypt(HCRYPTKEY hKey,
                         HCRYPTHASH hHash,
                         BOOL Final,
                         DWORD dwFlags,
                         BYTE *pbData,
                         DWORD *pdwDataLen,
                         DWORD dwBufLen);
```

В параметре *hKey* передается дескриптор ключа, необходимый для шифрования. Этот ключ также определяет алгоритм шифрования. Параметр *hHash* используется, если исходные данные одновременно шифруются и хэшируются. В нашем случае этот параметр следует установить равным нулю. Параметр *Final* следует установить в *TRUE*, если переданный в функцию блок данных является единственным или последним. В этом случае, он будет дополнен до необходимого размера. В противном случае передается значение *FALSE*. Параметр *dwFlags* не используется, и на его месте следует указать ноль. Параметр *pbData* – указатель на буфер, в котором содержатся данные для зашифрования. Зашифрованные данные по завершении работы функции помещаются в тот же буфер. Через параметр *pdwDataLen* в функцию при вызове передается размер исходных данных, а по завершении возвращается размер зашифрованных. *dwBufLen* – размер выходного буфера, для блочных шифров может быть больше, чем *pdwDataLen*.

Если используется блочный шифр AES, то максимальное значение параметра *pdwDataLen* при вызове функции составляет (*dwBufLen* - 16). Это связано с тем, что шифрование по умолчанию ведется в режиме CBC. Размер блока в данной реализации алгоритма всегда равен 128 битам. При зашифровании последний блок (*Final* устанавливается в *TRUE*) всегда конкатенируется с дополнением, размер которого в данном случае может составлять от 1 до 16 байт. Реальный размер зашифрованного блока, используемый для его записи в выходной файл, возвращается также через параметр *pdwDataLen*. Если размер исходных данных при вызове превышает 128 бит, то функция самостоятельно делит их на блоки требуемого алгоритмом размера. При этом размер любого входного блока, кроме последнего, должен быть кратен 16 байтам.

Пример вызова функции для зашифрования не последнего блока:

```
dwBuflen = 496;
CryptEncrypt(hKeyH, 0, FALSE,0, aBuf, &dwBuflen, 512);
```

CryptDecrypt

Функция осуществляет расшифрование данных. Прототип имеет вид:

```
BOOL WINAPI CryptDecrypt(HCRYPTKEY hKey,
                         HCRYPTHASH hHash,
                         BOOL Final,
                         DWORD dwFlags,
                         BYTE *pbData,
                         DWORD *pdwDataLen);
```

Все параметры функции имеют тоже значение, что и для функции ***CryptEncrypt***. Шифртекст передается через параметр *pbData*, а его длина через *pdwDataLen*. Функция возвращает реальную длину расшифрованного блока также через параметр *pdwDataLen*. Размер выходных данных, в отличие от шифрующей функции, не указывается. Это связано с тем, что размер расшифрованных данных может только уменьшаться и отведенного при вызове размера буфера будет вполне достаточно.

Пример вызова функции для расшифрования не последнего блока. Переменная *dwBuflen* должна содержать размер передаваемых через буфер *aBuf* данных:

```
CryptDecrypt(hKeyH, 0, FALSE, 0, aBuf, &dwBuflen);
```

CryptReleaseContext

После завершения работы с функциями CryptoAPI, необходимо освободить контекст криптопровайдера, полученный ранее с помощью функции ***CryptAcquireContext***. Для этого вызывается функция ***CryptReleaseContext***. Ее прототип имеет вид:

```
BOOL WINAPI CryptReleaseContext(HCRYPTPROV hProv,
                                DWORD dwFlags);
```

Параметр *hProv* – дескриптор освобождаемого контекста криптопровайдера. Параметр *dwFlags* должен равняться нулю. Пример вызова функции:

```
CryptReleaseContext(hProv, 0);
```

Интерфейс Cryptography API: Next Generation

Криптографический интерфейс CryptoAPI, частично описанный выше, поддерживался и продолжает поддерживаться всеми версиями операционных систем семейства Windows. Однако, начиная с версии Windows Vista среди настольных ОС и Windows 2008 Server среди

серверных, появилась поддержка так называемого криптографического интерфейса следующего поколения: *Cryptography API: Next Generation* (сокращенно *CNG*).

Сравнивая интерфейс CNG с предшественником можно сказать, что по многим параметрам они схожи, но есть и различия. В CryptoAPI 1.0 центральным понятием является криптопровайдер (CSP). Существует относительно большое число типов криптопровайдеров, распространяемых для ОС Windows. Пользователю необходимо получить контекст одного из криптопровайдеров нужного типа и использовать возвращенный ему дескриптор для получения доступа к его функциям.

В CNG также имеется понятие провайдера. Однако порядок действий с ними несколько иной. Пользователь непосредственно может получать дескриптор нужного ему криптоалгоритма, не задумываясь к какому типу относится предоставляющий его провайдер. По сути, в CNG осталось всего четыре типа провайдеров. В стандартную поставку Windows входят обычно четыре CNG-провайдера, каждый из которых относится к соответствующему типу:

- «Microsoft Primitive Provider»;
- «Microsoft Smart Card Key Storage Provider»;
- «Microsoft Software Key Storage Provider»;
- «Microsoft SSL Protocol Provider».

Программы, которые используют средства первого и последнего из перечисленных провайдеров могут работать как в режиме ядра, так и пользовательском режиме. Использование остальных провайдеров возможно только в пользовательском режиме. Сведения об имеющихся провайдерах CNG можно получить в ветви реестра *HKLM\SYSTEM\CurrentControlSet\Control\Cryptography*. В частности, там содержится информация о том, какой из провайдеров CNG поддерживает выполнение того или иного криптоалгоритма, а также информация о месторасположении провайдеров.

Из названий вышеперечисленных провайдеров становится ясным их назначение. «Microsoft Primitive Provider» поддерживает выполнение функций так называемых криптографических примитивов: генерации ключей и случайных чисел, шифрования, хэширования, экспорта и импорта ключей. Провайдер «Microsoft Software Key Storage Provider» отвечает за хранение ключей асимметричного шифрования и цифровой подписи. Если такие ключи должны храниться на смарт-картах, то используется «Microsoft Smart Card Key Storage Provider». Ну и, наконец, провайдер «Microsoft SSL

Protocol Provider» применяется в приложениях, реализующих протоколы безопасной передачи данных SSL и TLS.

Далее в данной работе будут рассматриваться особенности применения криптографических примитивов для симметричного шифрования данных. Все содержимое «Microsoft Primitive Provider» для работы в пользовательском режиме находится в файле *bcryptprimitives.dll*. По сути это набор провайдеров отдельных алгоритмов. «Microsoft Primitive Provider» поддерживает работу всех тех же алгоритмов, которые реализованы в криптопровайдерах CryptoAPI от Microsoft. Кроме того, добавлена поддержка реализации алгоритмов Диффи-Хеллмана и DSA с использованием эллиптических кривых (ECDH, ECDSA), а также новых алгоритмов генерации псевдослучайных чисел.

Для использования криптографических примитивов пользователь напрямую работает не с указанной выше библиотекой, а с файлом так называемого *маршрутизатора примитивов* (primitive router). Этот файл содержит функции, которые получают доступ к одному из криптографических интерфейсов: асимметричного, симметричного шифрования, хэширования и других, реализованных в библиотеке *bcryptprimitives.dll*. В качестве маршрутизатора при работе с примитивами в пользовательском режиме используется библиотека *bcrypt.dll*.

Соответственно, все функции, реализованные в данном файле, имеют префикс *BCrypt* (буква «B» означает «базовый). Например, ***BCryptOpenAlgorithmProvider*** или ***BCryptEncrypt***. Для доступа к сервисам других провайдеров в основном используются функции, реализованные в файле *ncrypt.dll*, и имеющие в своем названии префикс *NCrypt* (буква «N» означает «новый»). Также в данном файле реализованы функции для работы с «Microsoft SSL Protocol Provider» и имеющие, соответственно, префикс *Ssl*.

Рассмотрим более подробно, как выполняется симметричное шифрование данных с использованием криптографических примитивов CNG. Для этого необходимо выполнить следующие действия:

1. Загрузить и инициализировать провайдер нужного алгоритма шифрования.
2. Если необходимо, установить требуемые параметры алгоритма.
3. Определить размер ключевого объекта и выделить для него память.
4. Сгенерировать (или импортировать) сеансовый ключ.
5. Если необходимо, установить требуемые параметры ключа.

6. Зашифровать/расшифровать данные.
7. Экспортировать (если необходимо) и уничтожить ключ.
8. Закрыть провайдер алгоритма шифрования.

Генерацию ключа, также как и в CryptoAPI, можно выполнять с помощью генератора случайных чисел или на основе хэшированной парольной фразы. В данной лабораторной работе не рассматриваются вопросы экспорта и импорта ключей. Поэтому для генерации сеансового ключа будет применяться второй способ, из чего следует, что перед выполнением пункта 1, дополнительно должны быть выполнены следующие действия:

1. Загрузить и инициализировать провайдер нужного алгоритма хэширования.
2. Определить размер будущего хэш-объекта и выделить для него память.
3. Создать хэш-объект.
4. Поместить в хэш-объект парольную фразу.
5. Извлечь вычисленный хэш-код, который в дальнейшем будет передан функции, генерирующей сеансовый ключ.
6. Уничтожить хэш-объект.
7. Закрыть провайдер алгоритма хэширования.

Далее будут рассмотрены функции, которые потребуются для реализации приведенных выше алгоритмов. Все прототипы функций будут приведены на языке C/C++, как они и указаны в документации. В проектах Win32 для получения доступа к функциям работы с примитивами необходимо загрузить библиотеку *bcrypt.dll*. Для этого можно использовать библиотеку импорта *bcrypt.lib*, указав этот файл в свойствах проекта (*Проект→ Свойства → Свойства конфигурации → Компоновщик → Ввод → Дополнительные зависимости*) или указав строку в тексте программы:

```
#pragma comment(lib, " bcrypt.lib")
```

В файле исходного кода необходимо включить в текст директивой *#include* содержимое файлов *windows.h* и *bcrypt.h*. Причем включать файлы нужно именно в этом порядке, так как *bcrypt.h*, содержащий прототипы функций, константы, макросы, типы данных модуля BCrypt, использует в свою очередь некоторые типы данных, определенные в файлах, включение которых производится с помощью *windows.h*.

Все функции CNG, оперирующие строковыми параметрами, существуют только в одном варианте: с Unicode-строками. Поэтому строковые переменные нужно определять типом *WCHAR* (или *TCHAR*

и указывать в свойствах проекта, что используется Unicode), а перед константами ставить префикс “*L”* (или использовать макрос *TEXT*). Однако необходимость непосредственного оперирования ими возникает достаточно редко, так как для всех используемых функциями строковых констант в *bcrypt.h* определены символические константы.

Что касается специальных типов данных, определенных в *bcrypt.h*, то для описания приведенных ниже функций мы будем использовать переменные всего трех из них:

```
BCRYPT_ALG_HANDLE  hAlg;
BCRYPT_KEY_HANDLE hKey;
BCRYPT_HASH_HANDLE hHash;
```

Переменная *hAlg* будет использоваться в качестве дескриптора провайдера алгоритма, *hKey* в качестве дескриптора ключа, а *hHash* в качестве дескриптора хэш-объекта. Причем все три типа определены в файле *bcrypt.h* одинаково, как указатели на тип *VOID*. Что касается остальных типов параметров описываемых функций, то они являются стандартными типами Win32 API: *DWORD, PBYTE*.

Все функции CNG возвращают 32-битное значение типа *NTSTATUS*, которое представляет собой код ошибки ядра. Возможные значения определены в файле *ntstatus.h*. По структуре этот код схож со значением, возвращаемым функцией ***GetLastError***. Одним из вариантов определения успешности выполнения функции, является использование макроса *BCRYPT_SUCCESS*, определенного в файле *bcrypt.h*:

```
#define BCRYPT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)
```

Если значение, возвращаемое макросом *BCRYPT_SUCCESS* истинно, то функция считается выполненной успешно. Проверка на неотрицательность в макросе производится потому, что только при ошибке старший разряд кода устанавливается в единицу. А поскольку тип *NTSTATUS* определен как тип *LONG* (знаковый), то значение с единичным старшим битом будет восприниматься как отрицательное.

Описание большинства из представленных ниже функций является неполным, но достаточным для выполнения данной лабораторной работы. Полное описание этих и других функций можно найти в разделе MSDN, посвященном использованию CNG (http://msdn.microsoft.com/en-us/library/windows/desktop/aa376210%28v=vs.85%29.aspx).

BCryptOpenAlgorithmProvider

Данная функция загружает и инициализирует провайдер конкретного алгоритма. Ее прототип имеет вид:

```
NTSTATUS WINAPI BCryptOpenAlgorithmProvider(
                BCRYPT_ALG_HANDLE *phAlgorithm,
                LPCWSTR pszAlgId,
                LPCWSTR pszImplementation,
                DWORD dwFlags);
```

Параметр *phAlgorithm* представляет собой указатель на переменную, принимающую возвращаемый дескриптор провайдера указанного алгоритма. Идентификатор запрашиваемого алгоритма задает второй параметр *pszAlgId*. Он представляет собой указатель на Unicode-строку с нуль-символом в конце, содержащую зарегистрированное имя одного из алгоритмов. Для стандартного CNG-провайдера «Microsoft Primitive Provider» в файле *bcrypt.h* определены строковые константы с именами зарегистрированных алгоритмов и соответствующие им символические константы, которые, как правило, и указываются в качестве второго параметра. При выполнении данной работы для шифрования будет использоваться алгоритм AES, а для хэширования – SHA-256. Поэтому из всех определенных идентификаторов нам понадобятся только два. Их описание в файле *bcrypt.h* приведено ниже:

```
#define  BCRYPT_AES_ALGORITHM          L"AES"
#define  BCRYPT_SHA256_ALGORITHM       L"SHA256"
```

Параметр *pszImplementation* является указателем на Unicode-строку, содержащую зарегистрированное имя (псевдоним) одного из провайдеров криптографических примитивов. Если в качестве этого параметра указать константу *NULL*, то будет использован провайдер, определенный в реестре для данного алгоритма по умолчанию («Microsoft Primitive Provider»).

Параметр *dwFlags* для реализации обычного шифрования или хэширования должен быть равен нулю.

Ниже показан пример вызова функции для подключения к провайдеру алгоритма AES и один из возможных вариантов обработки возвращаемого функцией значения:

```
NTSTATUS status = STATUS_UNSUCCESSFUL;
if(!BCRYPT_SUCCESS(status = BCryptOpenAlgorithmProvider(
                            &hAlg,
                            BCRYPT_AES_ALGORITHM,
                            NULL, 0)))
```

```
{
     _tprintf(TEXT("Ошибка 0x%x получения\
              дескриптора алгоритма"), status);
}
```

Если необходимо подключиться к провайдеру алгоритма хэширования SHA-256, то используется идентификатор *BCRYPT_SHA256_ALGORITHM*.

Полученный через параметр *phAlgorithm* дескриптор алгоритма, в зависимости от его типа, в дальнейшем используется для создания либо ключевого объекта, либо хэш-объекта.

BCryptGetProperty

Функция позволяет получить значение одного из свойств объекта CNG (алгоритма, ключевого объекта, хэш-объекта) и имеет следующий прототип:

```
NTSTATUS WINAPI BCryptGetProperty(
                BCRYPT_HANDLE   hObject,
                LPCWSTR pszProperty,
                PUCHAR   pbOutput,
                ULONG   cbOutput,
                ULONG   *pcbResult,
                ULONG   dwFlags);
```

В качестве параметра *hObject* в функцию может передаваться дескриптор любого из объектов CNG, указанных выше. Параметр *pszProperty* является указателем на Unicode-строку, содержащую имя запрашиваемого свойства. Значения свойств могут быть разных типов, но все они возвращаются через параметр *pbOutput*. По сути функции передается адрес буфера в памяти, куда передается значение свойства. Поэтому передаваемый функции параметр должен приводиться к типу *PUCHAR* или *PBYTE*. Размер буфера задается параметром *cbOutput*, а параметр *pcbResult* задает адрес переменной, куда возвращается размер данных в байтах, реально скопированных в буфер *pbOutput*. Параметр *dwFlags* должен быть равен нулю.

Данная функция совместно с функцией ***BCryptSetProperty*** в качестве идентификаторов свойств может использовать символические константы, определенные в файле *bcrypt.h*. Некоторые из них приведены ниже:

– *BCRYPT_ALGORITHM_NAME*. Позволяет определить имя используемого алгоритма;

- *BCRYPT_BLOCK_LENGTH*. Позволяет получить размер блока алгоритма шифрования в байтах;
- *BCRYPT_CHAINING_MODE*. Позволяет получить название установленного режима шифрования. Для основных режимов блочного шифрования определены символические константы: *BCRYPT_CHAIN_MODE_CBC*, *BCRYPT_CHAIN_MODE_CFB*, *BCRYPT_CHAIN_MODE_ECB*, которые соответствуют строковым константам, содержащим названия режимов;
- *BCRYPT_KEY_LENGTH*. Позволяет получить размер ключа в битах;
- *BCRYPT_HASH_LENGTH*. Позволяет получить размер хэш-кода загруженного алгоритма хэширования;
- *BCRYPT_OBJECT_LENGTH*. Используется для получения размера в байтах объектов провайдера некоторых типов.

Что касается последнего идентификатора, то он используется при вызове функции, в частности, если нужно определить размер ключевого объекта перед созданием сеансового ключа или размер хэш-объекта. В приведенном ниже примере определяется размер объекта, тип которого зависит от типа загруженного алгоритма. Если переменная *hAlg* содержит дескриптор алгоритма шифрования, то через параметр *pbOutput* возвратится размер ключевого объекта, а если загружен алгоритм хэширования, то вернется размер хэш-объекта. В приведенном ниже примере и во всех последующих для краткости не записывается «обвязка» вызова, проверяющая возвращаемое функцией значение.

```
DWORD cbObject=0, cbData=0;
PBYTE pbObject=NULL;
BCryptGetProperty(hAlg, BCRYPT_OBJECT_LENGTH,
                  (PBYTE)&cbObject, sizeof(DWORD),
                  &cbData, 0);
//выделение памяти для объекта:
pbObject=(PBYTE)HeapAlloc(GetProcessHeap(), 0,
                          cbObject);
//выполнение каких-либо операций с объектом
.....................................
//освобождение  объекта
HeapFree(GetProcessHeap(), 0, pbObject);
```

Далее созданный в памяти буфер может быть передан функциям, которые разместят в нем, соответственно, или ключевой объект, или хэш-объект. В переменную *cbData* в этом случае будет записан размер в байтах переданных в *cbObject* данных, то есть 4.

BCryptSetProperty

Функция позволяет установить значение одного из именованных свойств объекта CNG (алгоритма, ключевого объекта, хэш-объекта). Ее прототип во многом повторяет прототип предыдущей функции:

```
NTSTATUS WINAPI BCryptSetProperty(
                BCRYPT_HANDLE   hObject,
                LPCWSTR pszProperty,
                PUCHAR   pbInput,
                ULONG   cbInput,
                ULONG   dwFlags);
```

Разница между этой функцией и предыдущей заключается в том, что через параметр *pbInput* передается новое значение свойства, заданного параметром *pszProperty*. Через параметр *cbInput* передается размер этого значения. Параметр dwFlags также должен иметь значение нуля.

Формально функция ***BCryptSetProperty*** может использовать те же идентификаторы, что и функция ***BCryptGetProperty***. Однако из всех перечисленных в описании предыдущей функции идентификаторов, функция ***BCryptSetProperty*** может использовать только один: *BCRYPT_CHAINING_MODE*. Другие рассмотренные идентификаторы определяют свойства, доступные только для чтения. Ниже показан пример вызова функции, устанавливающий для полученного ранее дескриптора алгоритма симметричного шифрования режим CFB:

```
BCryptSetProperty(hAlg, BCRYPT_CHAINING_MODE,
                  (PBYTE)BCRYPT_CHAIN_MODE_CFB,
                  sizeof(BCRYPT_CHAIN_MODE_CFB), 0);
```

BCryptCreateHash

Функция создает хэш-объект и имеет следующий прототип:

```
NTSTATUS WINAPI BCryptCreateHash(
                BCRYPT_ALG_HANDLE hAlgorithm,
                BCRYPT_HASH_HANDLE *phHash,
                PUCHAR pbHashObject,
                ULONG cbHashObject,
                PUCHAR pbSecret,
                ULONG cbSecret,
                ULONG dwFlags);
```

Параметр *hAlgorithm* представляет полученный ранее с помощью функции ***BCryptOpenAlgorithmProvider*** дескриптор алгоритма хэширования. Параметр *phHash* является указателем на переменную типа *BCRYPT_HASH_HANDLE*, которая будет содержать дескриптор

создаваемого хэш-объекта. В дальнейшем этот дескриптор будет передаваться функциям, которые непосредственно выполняют хэширование данных и выгрузку хэш-кода.

Параметр *pbHashObject* содержит указатель на буфер, в котором размещается созданный хэш-объект. Способ получения этого буфера был описан ранее. Параметр *cbHashObject* содержит размер этого буфера. Параметр *pbSecret* передает адрес буфера, содержащего ключ, используемый для создания кода аутентичности сообщения (MAC). Поскольку мы будем выполнять обычное хэширование, то вместо этого параметра нужно указывать константу *NULL*, а в качестве значения параметра *cbSecret* – 0. Параметр *dwFlags* в нашем случае тоже должен быть равен нулю.

Ниже показан пример вызова этой функции. Предполагается, что буфер, адресуемый указателем *pbHashObject* и его размер *cbHashObject* получены заранее:

```
BCryptCreateHash(hAlg, &hHash, pbHashObject,
                 cbHashObject, NULL, 0, 0);
```

BCryptHashData

Функция непосредственно выполняет хэширование передаваемых ей данных, используя созданный ранее с помощью функции ***BCryptCreateHash*** хэш-объект. Прототип функции имеет следующий вид:

```
NTSTATUS WINAPI BCryptHashData(
                BCRYPT_HASH_HANDLE hHash,
                PUCHAR pbInput,
                ULONG cbInput,
                ULONG dwFlags);
```

Параметр *hHash* представляет собой дескриптор существующего хэш-объекта. Параметр *pbInput* содержит адрес буфера с входными данными, а параметр *cbInput* его размер. Параметр *dwFlags* должен быть равен нулю. Функция может вызываться многократно для одного и того же хэш-объекта, добавляя в него новые данные для хэширования.

Ниже показан пример хэширования пароля, который находится в строке *szPass*.

```
TCHAR szPass[100];
.............................
//Получение каким-либо образом пароля и занесение его в szPass.
.............................
BCryptHashData(hHash,(PBYTE)szPass,
               _tcslen(szPass)*sizeof(TCHAR),0);
```

BCryptFinishHash

Функция извлекает из хэш-объекта окончательный хэш-код входных данных, полученный с помощью одного или нескольких вызовов функции ***BCryptHashData.*** Прототип функции имеет вид:

```
NTSTATUS WINAPI BCryptFinishHash(
                BCRYPT_HASH_HANDLE hHash,
                PUCHAR pbOutput,
                ULONG cbOutput,
                ULONG dwFlags);
```

Параметр *hHash* является дескриптором хэш-объекта. Буфер, заданный указателем *pbOutput*, будет принимать выгружаемый хэш-код и должен быть создан заранее. Параметр *cbOutput* содержит размер этого буфера. Параметр *dwFlags* должен быть равен нулю.

Ниже показан пример выгрузки хэш-кода:

```
PBYTE pbHash= NULL;
DWORD cbData= 0, cbHash= 0;
BCryptGetProperty(hAlg, BCRYPT_HASH_LENGTH,
                  (PBYTE)&cbHash, sizeof(DWORD),
                  &cbData, 0);
pbHash= (PBYTE)HeapAlloc(GetProcessHeap(), 0, cbHash);
BCryptFinishHash(hHash, pbHash, cbHash, 0);
```

BCryptGenerateSymmetricKey

Функция создает ключевой объект и помещает в него ключ симметричного шифрования загруженного ранее алгоритма. Ключ создается на основе ключевого материала, передаваемого функции. Ее прототип имеет вид:

```
NTSTATUS WINAPI BCryptGenerateSymmetricKey(
                BCRYPT_ALG_HANDLE hAlgorithm,
                BCRYPT_KEY_HANDLE *phKey,
                PUCHAR pbKeyObject,
                ULONG cbKeyObject,
                PUCHAR pbSecret,
                ULONG cbSecret,
                ULONG dwFlags);
```

Параметр *hAlgorithm* представляет полученный ранее с помощью функции ***BCryptOpenAlgorithmProvider*** дескриптор алгоритма шифрования. Параметр *phKey* является указателем на переменную типа *BCRYPT_KEY_HANDLE*, которая будет содержать дескриптор создаваемого ключевого объекта. Параметр *pbKeyObject* содержит указатель на буфер, в котором размещается созданный ключевой

объект. Способ получения этого буфера был описан ранее. Параметр *cbKeyObject* содержит размер данного буфера в байтах.

Параметр *pbSecret* является указателем на буфер, который содержит ключевой материал. По сути, он и будет являться ключом. Получить ключевой материал можно двумя способами: генерацией псевдослучайного числа требуемого размера или созданием хэш-кода парольной фразы. Поэтому в нашем случае в качестве этого параметра функции будет передаваться указатель на буфер с хэш-кодом, извлеченным ранее функцией ***BCryptFinishHash***. Через параметр *cbSecret* передается размер в байтах буфера с ключевым материалом.

Если вспомнить генерацию сеансового ключа AES в CryptoAPI, то там использовались отдельные идентификаторы алгоритма для разных длин ключа. Здесь же длина генерируемого ключа фактически определяется значением, передаваемым через параметр *cbSecret*. Если, к примеру, через параметр *pbSecret* передается адрес буфера, содержащего 32-байтный (256-битный) хэш-код, то задавая через параметр *cbSecret* значения 16, 24 или 32 мы получим ключ длиной, соответственно, 128, 192 или 256 бит.

Параметр *dwFlags* должен быть равен нулю. Ниже показан пример вызова функции для создания 128-битного ключа. Предполагается, что на момент вызова функции уже определены значения параметров *pbKeyObject* (содержит адрес буфера для размещения ключевого объекта), *cbKeyObject* (содержит размер буфера в байтах) и *pbHash* (содержит адрес буфера с хэш-кодом).

```
BCryptGenerateSymmetricKey(hAlg, &hKey, pbKeyObject,
                           cbKeyObject, (PBYTE)pbHash,
                           16, 0);
```

Для созданного после вызова функции дескриптора ключа можно определять и устанавливать некоторые параметры с помощью функций ***BCryptGetProperty*** и ***BCryptSetProperty***. Например, если установлен режим шифрования CFB, то по умолчанию он используется в 8-битном варианте. Для того чтобы установить иной размер обратной связи (в байтах) можно использовать функцию ***BCryptSetProperty*** с идентификатором *BCRYPT_MESSAGE_BLOCK_LENGTH*.

После создания ключа и установления его параметров (если это необходимо) можно передавать его дескриптор функциям зашифрования или расшифрования.

BCryptEncrypt

Функция зашифровывает переданный ей блок данных произвольного размера. Ее прототип имеет вид:

```
NTSTATUS WINAPI BCryptEncrypt(
                BCRYPT_KEY_HANDLE hKey,
                PUCHAR pbInput,
                ULONG cbInput,
                VOID *pPaddingInfo,
                PUCHAR pbIV,
                ULONG cbIV,
                PUCHAR pbOutput,
                ULONG cbOutput,
                ULONG *pcbResult,
                ULONG dwFlags);
```

Через параметр *hKey* передается дескриптор созданного ранее ключа. Параметр *pbInput* содержит адрес буфера с входными данными (открытым текстом). Параметр *cbInput* содержит размер входных данных в байтах. Параметр *pPaddingInfo* может содержать указатель на одну из двух возможных структур, определяющих параметры дополнения блоков при шифровании. В случае обычного симметричного шифрования этот параметр не используется и должен быть равен *NULL*.

Через параметр *pbIV* можно передавать адрес буфера, содержащего вектор инициализации (синхропосылку). Обычно он получается с помощью генератора псевдослучайных чисел. Размер вектора в байтах передается через параметр *cbIV* и он должен совпадать с размером блока. То есть если размер блока алгоритма AES равен 16 байтам (128 бит), то вектор инициализации должен иметь такой же размер. Также нужно учесть, что в процессе зашифрования функция изменяет содержимое буфера с вектором. Поэтому если в дальнейшем вновь понадобится использовать его значение, то перед вызовом функции необходимо создать копию вектора. Если при зашифровании используется нулевой вектор инициализации, то параметры *pbIV* и *cbIV* должны быть заданы, соответственно, как *NULL* и 0.

Параметр *pbOutput* содержит адрес буфера, в который будет помещаться шифртекст. Размер буфера в байтах передается через параметр *cbOutput*. Если в качестве параметра *pbOutput* передать *NULL*, то через параметр *pcbResult* будет возвращен размер выходного буфера, необходимого для размещения шифртекста, который получится из указанных входных данных. Если же параметр *pbOutput*

при вызове содержит адрес выходного буфера, то через *pcbResult* возвращается количество байт, реально помещенных в этот буфер.

Параметр *dwFlags* в случае симметричного шифрования может быть равен нулю или константе *BCRYPT_BLOCK_PADDING*, которая определяет необходимость дополнения последнего блока. Флаг устанавливается в ноль, если четко известно, что размер входных данных кратен размеру блока алгоритма симметричного шифрования. Если же при нулевом флаге размер открытого текста не кратен размеру блока, то восстановить его в дальнейшем может не получиться, вне зависимости от выбранного режима шифрования. Поэтому при шифровании файлов флаг всегда нужно устанавливать равным *BCRYPT_BLOCK_PADDING*, независимо от выбранного режима шифрования (в том числе и при выборе режима CFB). Следовательно, размер выходных данных будет всегда больше (для AES на величину от 1 до 16 байт), чем входных.

Отсюда следует еще один вывод. Параметры *pbInput* и *pbOutput* могут содержать адрес одного и того же буфера. Однако этот буфер нельзя полностью заполнять входными данными, так как заменяющие их по мере зашифрования выходные данные будут иметь больший размер (для AES максимально на 16 байт).

Ниже показан пример зашифрования массива однобайтных величин с нулевым вектором инициализации и предварительным определением размера буфера для размещения шифртекста:

```
BYTE Plaintext[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
DWORD cbOutput=0, cbCiphertext=0;
PBYTE pbOutput=NULL;
BCryptEncrypt(hKey, Plaintext, sizeof(Plaintext),
              NULL, NULL, 0, NULL, 0, &cbOutput,
              BCRYPT_BLOCK_PADDING);
pbOutput=(PBYTE)HeapAlloc(GetProcessHeap(), 0,
                          cbOutput);
BCryptEncrypt(hKey, Plaintext, sizeof(Plaintext),
              NULL, NULL, 0, pbOutput, cbOutput,
              &cbCiphertext, BCRYPT_BLOCK_PADDING);
```

BCryptDecrypt

Функция расшифровывает переданный ей блок данных произвольного размера. Ее прототип имеет вид:

```
NTSTATUS WINAPI BCryptDecrypt(
                BCRYPT_KEY_HANDLE hKey,
                PUCHAR pbInput,
                ULONG cbInput,
                VOID *pPaddingInfo,
```

```
                PUCHAR pbIV,
                ULONG cbIV,
                PUCHAR pbOutput,
                ULONG cbOutput,
                ULONG *pcbResult,
                ULONG dwFlags);
```

Все параметры этой функции по назначению аналогичны параметрам функции ***BCryptEncrypt***. Отличие состоит лишь в том, что через параметр *pbInput* передается адрес буфера с шифртекстом, а параметр *pbOutput* указывает на буфер, в котором будет размещен восстановленный открытый текст. Рассмотренный ранее способ определения размера выходного буфера может применяться и для этой функции. Параметр *dwFlags* также должен иметь значение *BCRYPT_BLOCK_PADDING*.

BCryptDestroyHash

Функция уничтожает созданный ранее хэш-объект и имеет следующий прототип:

```
NTSTATUS WINAPI BCryptDestroyHash(
                BCRYPT_HASH_HANDLE hHash);
```

Единственный параметр представляет собой дескриптор уничтожаемого хэш-объекта.

BCryptDestroyKey

Функция уничтожает созданный ранее ключевой объект и имеет следующий прототип:

```
NTSTATUS WINAPI BCryptDestroyKey(
                BCRYPT_KEY_HANDLE hKey);
```

В качестве параметра функция принимает дескриптор уничтожаемого ключа.

BCryptCloseAlgorithmProvider

Функция выгружает загруженный ранее провайдер алгоритма и имеет следующий прототип:

```
NTSTATUS WINAPI BCryptCloseAlgorithmProvider(
                BCRYPT_ALG_HANDLE hAlgorithm,
                ULONG dwFlags);
```

Параметрами функции являются дескриптор провайдера и флаг, который должен быть равен нулю.

СОДЕРЖАНИЕ РАБОТЫ

Разработать на языке программирования C/C++ консольное или оконное приложение, выполняющее зашифрование и расшифрование файла произвольного формата с помощью алгоритма AES-128, AES-192 или AES-256 по выбору пользователя. Программа должна генерировать сеансовый ключ на основании хэшированного пароля, который запрашивается у пользователя. Использовать хэш-функцию SHA-256.

Приложение должно по выбору пользователя использовать функционал двух криптографических интерфейсов:

- CryptoAPI (в этом случае режим блочного шифрования – устанавливаемый по умолчанию CBC с нулевым вектором инициализации, используемый криптопровайдер – «Microsoft Enhanced RSA and AES Cryptographic Provider»).
- Cryptography API: Next Generation (дополнительно предоставляется выбор режим блочного шифрования: ECB, CBC или CFB, причем в последних двух также используется нулевой вектор инициализации).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое CryptoAPI? В чем заключается различие между CryptoAPI 1.0 и CryptoAPI 2.0?
2. Что такое криптопровайдер? Как можно подключиться к криптопровайдеру?
3. Какое количество функций должен поддерживать криптопровайдер?
4. Как создать контейнер ключей? Какие типы ключей в нем будут храниться?
5. Какие типы криптопровайдеров вы знаете? Чем они различаются?
6. Как можно выполнить генерацию ключа симметричного шифрования?
7. Какой режим шифрования устанавливается при генерации ключа по умолчанию?
8. Что такое хэш-объект? Какие функции для работы с хэш-объектами вы знаете?
9. Какие функции CryptoAPI выполняют зашифрование и расшифрование данных? Какие они имеют параметры?

10. Что такое Cryptography API: Next Generation? В чем заключаются его различия с CryptoAPI?
11. Какие типы провайдеров CNG доступны в операционных системах Windows? Как можно узнать, какие конкретно провайдеры установлены в системе?
12. Средства каких провайдеров CNG можно использовать в режиме ядра?
13. Какие алгоритмы поддерживает провайдер криптографических примитивов Microsoft?
14. Что такое маршрутизатор примитивов?
15. Как определить успешность вызова функции CNG?
16. Как загрузить и выгрузить провайдер нужного алгоритма?
17. Как создается хэш-объект? Какие функции CNG для работы с хэш-объектами вы знаете?
18. Как определить размер буфера при выгрузке хэш-кода?
19. Как сгенерировать ключ симметричного шифрования и установить его параметры?
20. Какие функции CNG выполняют зашифрование и расшифрование данных? Какие они имеют параметры?

ЛАБОРАТОРНАЯ РАБОТА № 4

СИММЕТРИЧНОЕ И АСИММЕТРИЧНОЕ ШИФРОВАНИЕ ДАННЫХ СРЕДСТВАМИ КРИПТОГРАФИЧЕСКОГО ПАКЕТА OPENSSL

Цель работы: ознакомиться со средствами симметричного и асимметричного шифрования, предоставляемыми пакетом OpenSSL и способами доступа к ним из приложений на языке C/C++.

ОСНОВНЫЕ ПОНЯТИЯ

Криптографический пакет OpenSSL

Пакет предназначен прежде всего для обеспечения работы веб-серверов, поддерживающих передачу данных через защищенные протоколы SSL (Secure Sockets Layer) и TLS (Transport Layer Security). В частности, он позволяет генерировать ключевые пары различных асимметричных алгоритмов, сеансовые ключи блочных шифров, выполнять шифрование, создавать электронные подписи и т.д.

Взаимодействие с OpenSSL возможно в двух вариантах: через командную строку в режиме консольного приложения или непосредственное использование в приложении функций динамических библиотек, поставляемых в составе пакета. В этой работе будем использовать второй вариант, непосредственно вызывая нужные нам функции в тексте программы на языке C/C++. Далее вкратце опишем процесс инсталляции и первоначальной настройки пакета, а также порядок использования его библиотек в приложении Win32, создаваемом в среде Visual Studio.

Как правило, OpenSSL используется совместно с Unix-подобными системами, но как кроссплатформенное приложение может применяться и в ОС Windows. Документация, а также последняя версия пакета доступны по адресу: https://www.openssl.org. Загруженный пакет необходимо скомпилировать и сконфигурировать для работы в конкретной ОС. Для того чтобы избавить пользователя Windows от выполнения большинства этих действий существуют готовые дистрибутивы для 32 и 64-разрядных версий системы, доступные по адресу: http://slproweb.com/products/Win32OpenSSL.html. Кроме дистрибутива OpenSSL, необходимо с этой же страницы скачать и предварительно установить в системе пакет *Visual C++ 2008*

Redistributables (соответственно разрядности ОС). После его установки, необходимо запустить дистрибутив самого OpenSSL и далее идет процесс обычной инсталляции Windows-приложения с указанием пользователем места расположения копируемых файлов. Поскольку в среде Visual Studio будет создаваться 32-разрядное приложение, то рекомендуется скачивать и устанавливать 32-разрядную версию пакета OpenSSL независимо от разрядности ОС, так как в противном случае разрядность приложения и установленных динамических библиотек будет различаться и проект не соберется.

После установки пакет в принципе готов к работе, однако необходимо осуществить еще некоторые действия по его конфигурации. При рассмотрении всех примеров будем считать, что пакет был установлен в каталоге *C:\OpenSSL-Win32*. Непосредственно для работы с командной строки используется файл *openssl.exe*, расположенный в подкаталоге *bin*. Для проверки работоспособности пакета запустим этот файл на выполнение.

В консоли появится предупреждение и приглашение для ввода команд:

```
WARNING: can't open config file: /usr/local/ssl/openssl.cfg
OpenSSL>
```

Предупреждение означает, что при запуске не найден конфигурационный файл *openssl.cfg*. Приложение без этого файла остается работоспособным, что подтверждается выдачей приглашения. Однако в этом файле указывается ряд параметров, которые могут понадобиться в процессе эксплуатации пакета. Например, в нем указываются параметры подключения модуля *gost*, который реализует отечественные алгоритмы шифрования, хэширования и электронной подписи. Модуль с реализацией отечественных стандартов криптографии, появившийся в версии 1.0.0, был разработан компанией «Криптоком». Такой модуль присутствует по умолчанию в дистрибутивах, скачанных на ресурсе http://slproweb.com. Для его включения пока завершим работу OpenSSL, введя команду *exit* (или *quit*).

С помощью Проводника откроем каталог, где по умолчанию находится конфигурационный файл – *C:\OpenSSL-Win32\bin*. С помощью любого текстового редактора откроем файл *openssl.cfg*. Этот файл делится на ряд секций, названия которых указываются в квадратных скобках. Пока не будем вносить в них изменений, а добавим строки, включающие поддержку алгоритмов ГОСТ.

В качестве первой строки файла добавим указание на первую секцию настройки параметров подключения модуля:

```
openssl_conf = openssl_def
```

Секцию **[openssl_def]** и другие секции настройки параметров подключения модуля gost разместим в конце файла:

```
[openssl_def]
engines=engine_section

[engine_section]
gost=gost_section

[gost_section]
engine_id=gost
dynamic_path = C:/OpenSSL-Win32/bin/gost.dll
default_algorithms=ALL
CRYPT_PARAMS=id-Gost28147-89-CryptoPro-A-ParamSet
```

В последней секции указывается имя модуля, по которому к нему в дальнейшем можно обращаться в программе (*gost*), путь к *dll*-файлу модуля gost, включаемые алгоритмы (все реализованные) и параметр, являющийся идентификатором варианта таблицы замен алгоритма ГОСТ 28147-89, представленном в документе RFC 4357.

Выполнив все изменения, сохраним файл и закроем его. Теперь можно создать переменную среды, которая укажет для пакета путь доступа к конфигурационному файлу. Ее можно создать средствами Проводника (*Свойства системы* → *Дополнительные параметры системы* → *Переменные среды*) или командой в окне командной строки:

```
set OPENSSL_CONF=c:\openssl-win32\bin\openssl.cfg
```

Вновь запустим на выполнение файл *openssl.exe* и убедимся, что предупреждение не появляется. Проверим, появилась ли поддержка отечественных криптоалгоритмов. Для этого после приглашения введем команду:

```
ciphers -v
```

и убедимся, что среди представленных наборов алгоритмов (*шифр-сьютов*) есть такие, в которых фигурируют названия GOST-89, GOST-94, GOST2001 (соответственно ГОСТ 28147-89, ГОСТ Р 34.10-94, ГОСТ Р 34.10-2001). Закроем окно консоли.

Настроенная загрузка алгоритмов ГОСТ понадобится позже, а пока рассмотрим, как производится симметричное и асимметричное

шифрование средствами стандартного модуля пакета при доступе к его функциям из программы на языке C/C++.

В стандартную реализацию пакета входит большое число симметричных шифров (с возможностью работы в различных режимах) и несколько асимметричных алгоритмов шифрования, обмена ключами и электронной подписи (ЭП). В данной работе ограничимся использованием алгоритма AES для симметричного шифрования и RSA для асимметричного (имитации процесса обмен сеансовым ключом). Далее будут рассмотрен минимально необходимый для выполнения данной работы набор функций.

Инициализация библиотеки шифрования

Для начала посмотрим на содержимое каталога, в который установлен пакет OpenSSL. Из того, что в нем есть, нам понадобится:

- *dll*-файлы динамических библиотек, которые находятся в подкаталоге *bin*. В данной работе будут использоваться функции, входящие в состав библиотек *libeay32.dll* и *gost.dll*.
- *lib*-файлы статических библиотек (используются как библиотеки импорта подключаемых *dll*-библиотек), которые находятся в подкаталоге *lib*. В этой работе нам понадобятся библиотеки *libeay32.lib* и *ssleay32.lib*. Их необходимо подключить, например в свойствах проекта VC++ (*Проект* → *Свойства* → *Свойства конфигурации* → *Компоновщик* → *Ввод* → *Дополнительные зависимости*).
- Заголовочные файлы, содержащие прототипы функций, типы данных, константы, и которые находятся в подкаталоге *include*. Необходимые заголовочные файлы включаются в текст программы на языке C/C++ с помощью директивы препроцессора *#include*. В настройках проекта также нужно добавить каталог *C:\OpenSSL-Win32\include* (в нашем случае) в раздел каталогов VC++ (*Проект* → *Свойства* → *Свойства конфигурации* → *Каталоги VC++* → *Каталоги включения*).

В принципе, вызывать большинство функций, зная их сигнатуру (если подключить заранее нужный заголовочный файл, то ее подскажет сама IDE) можно без какой-то особой инициализации. Однако мы в качестве этой процедуры выполним в начале программы вызов двух функций, описанных ниже.

Функция ***OpenSSL_add_all_algorithms*** загружает во внутренние таблицы библиотеки все имеющиеся в стандартном модуле реализации

алгоритмы шифрования и хэширования. Прототип функции содержится в файле *openssl/evp.h* и имеет вид:

```
void OpenSSL_add_all_algorithms(void);
```

Если в процессе выполнения функций OpenSSL происходит ошибка, то для просмотра ее текстового описания нужно, чтобы была инициализирована внутренняя таблица сообщений об ошибках. Это делается с помощью функции ***ERR_load_error_strings***. Она объявлена в файле *openssl/err.h*:

```
void ERR_load_crypto_strings(void);
```

Далее рассмотрим, какие средства ввода-вывода библиотеки OpenSSL нам понадобятся в процессе выгрузки информации в файл и ее загрузки из файла.

Абстракция ввода-вывода BIO

В OpenSSL имеется специальный тип для представления разных видов источников и получателей данных – ***BIO***. Он скрывает от пользователя детали операций ввода-вывода, позволяя использовать для их реализации некоторый набор функций. Имеется два основных типа объектов *BIO*: источник/получатель данных или фильтр (конструкция, позволяющая связывать несколько структур *BIO* и осуществлять некоторые преобразования данных при передаче их из одной структуры в другую). Источник или получатель данных могут представлять собой файл, сокет или просто буфер в оперативной памяти. Независимо от того, к какому типу будет принадлежать создаваемый объект *BIO*, в программе переменная для работы с ним задается как указатель на одноименный тип *BIO*, определенный в файле *openssl/bio.h*. Тип *BIO* получен переименованием структурного типа *bio_st*, определенного в этом же файле и содержащего ряд полей, задающих информацию об объекте. Содержание структуры можно посмотреть в заголовочном файле.

В данной работе мы будем использовать только файловый *BIO*. Он создан на базе типа *FILE* стандартной библиотеки ввода-вывода языка Си. Создать переменную такого типа можно с помощью функции ***BIO_new_file***, объявленную в файле *openssl/bio.h*.

```
BIO *BIO_new_file(const char *filename, const char *mode);
```

Параметры функции аналогичны таковым у функции ***fopen***. Пример вызова:

```
BIO *fbio;
fbio = BIO_new_file("outfile.txt", "wb");
```

Для выполнения операций чтения-записи данных в файловом *BIO* имеется ряд функций, из которых мы рассмотрим только две.

BIO_read пытается прочитать заданное число байт структуры *BIO*. Имеет прототип:

```
int BIO_read(BIO *b, void *data, int len);
```

Параметр *len* определяет количество байт, которые функция пытается прочесть из объекта *b* и поместить их в буфер *data*. Функция возвращает количество реально считанных байт данных.

BIO_write пытается записать *len* байт буфера *data* в *b*. Имеет прототип:

```
int BIO_write(BIO *b, const void *data, int len);
```

Для удаления объекта *BIO*, независимо от его типа, используется функция ***BIO_free***.

```
void BIO_free(BIO *b);
```

С другими функциями для работы с объектами *BIO* можно ознакомиться в документации к пакету OpenSSL.

Генерация псевдослучайных чисел

В симметричном шифровании псевдослучайные числа используются для генерации секретного ключа и вектора инициализации (синхропосылки) для режимов, отличных от ECB. Для генерации псевдослучайных чисел используются две функции: ***RAND_bytes*** и ***RAND_pseudo_bytes***, прототипы которых объявлены в файле *openssl/rand.h*.

```
int RAND_bytes(unsigned char *buf, int num);
int RAND_pseudo_bytes(unsigned char *buf, int num);
```

Первая функция генерирует *num* криптографически сильных псевдослучайных чисел в буфер *buf*. Результат ее работы можно использовать в качестве сеансовых ключей симметричных криптоалгоритмов. Функция возвращает 1 в случае успеха и 0 в случае неудачи.

Функция ***RAND_pseudo_bytes*** действует аналогично, но достаточные статистические характеристики чисел не гарантируется. В случае если полученные псевдослучайные данные

криптографически сильные, возвращает 1, в случае если они недостаточно сильные, 0 и -1 в случае ошибки. Результат работы данной функции можно использовать в качестве вектора инициализации.

Однако перед тем как генерировать псевдослучайные числа, желательно инициализировать генератор, внеся в его параметры источник энтропии (*seed*). Простейшим вариантом может быть вызов функции ***RAND_screen***, которая берет случайные значения из хэша данных, полученных из скриншота содержимого экрана.

Более предпочтительным вариантом является непосредственное указание случайных данных с помощью функции ***RAND_add***.

```
void RAND_add(const void *buf, int num, double entropy);
```

Случайные величины в количестве *num* байт берутся из буфера *buf*. Параметр *entropy* задает энтропию сообщения, содержащегося в буфере. В общем случае можно задавать этот параметр равным *num*.

С другими функциями, объявленными в файле *openssl/rand.h* можно ознакомиться в документации к библиотеке.

Симметричное шифрование данных с помощью криптоалгоритма AES

Для симметричного шифрования будем использовать высокоуровневые функции, имеющие в своем названии префикс «EVP» и объявленные в файле *openssl/evp.h*.

Независимо от того, какой конкретно блочный шифр применяется, используется одинаковый набор функций для инициализации процесса шифрования, загрузки данных и завершения процесса. Эти функции используют две структуры, определенные в этом же заголовочном файле: *EVP_CIPHER* и *EVP_CIPHER_CTX*.

Первая структура содержит информацию о блочном шифре: размер блока, длина ключа и т.п., а также ряд указателей на функции, которые нужно инициализировать. Для этого существует несколько способов. Мы будем использовать непосредственный вызов функций, возвращающих константный указатель на динамически созданную структуру *EVP_CIPHER*, в названии которых содержатся данные об алгоритме, длине ключа и режиме шифрования. В данной работе ограничимся шифрованием AES со 128-разрядным ключом в режимах CBC и 128-разрядный CFB (размер блока в любых режимах всегда равен 128 бит). Соответственно это будут функции:

```
const EVP_CIPHER *EVP_aes_128_cbc(void);
```

```
const EVP_CIPHER *EVP_aes_128_cfb128(void);
```

Структура *EVP_CIPHER_CTX* представляет собой контекст алгоритма шифрования, первоначальную инициализацию которого осуществляют с помощью функции ***EVP_CIPHER_CTX_init.***

```
void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
```

Данная функция вызывается, если переменная типа *EVP_CIPHER_CTX* описана статически. Если она описана как указатель, то вызывается функция ***EVP_CIPHER_CTX_new***, которая выделяет под нее память и возвращает адрес структуры.

```
EVP_CIPHER_CTX *EVP_CIPHER_CTX_new();
```

В дальнейшем освобождает контекст алгоритма шифрования либо функция:

```
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
```

в случае статического размещения структуры, либо функция:

```
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *a);
```

После того, как инициализирован контекст алгоритма, можно вызвать функцию ***EVP_EncryptInit***, которая подготавливает его к выполнению операций шифрования:

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx,
                    const EVP_CIPHER *cipher,
                    const unsigned char *key,
                    const unsigned char *iv);
```

Параметры *key* и *iv* представляют собой буферы, содержащие ключевой материал и вектор инициализации, которые получаются с помощью функций генерации псевдослучайных чисел. Функция, как и большинство других, имеющих тип *int*, возвращает 1 в случае успеха и 0 в случае неудачи. Если вызов закончился неудачно, то можно вывести текстовое сообщение об ошибке. Ниже будет показан пример инициализации контекста, где вариант с ошибкой обрабатывается подобным образом. Такой подход можно использовать и при вызове других функций.

```
unsigned char keybuf[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                             10, 11, 12, 13, 14, 15 };
unsigned char iv[16] = { 0 },
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
```

```
int ret = EVP_EncryptInit(&ctx, EVP_aes_128_cbc(), keybuf,
                          iv);
if (!ret)
{
    char buffer[500];
    ERR_error_string(ERR_get_error(), buffer);
    printf("%s\n",buffer);
}
```

В этом примере значения ключа и вектора инициализации для простоты заданы непосредственно, но на практике для их получения нужно использовать псевдослучайные числа. Для получения кода ошибки из очереди была использована функция ***ERR_get_error***, а для получения текстового описания функция ***ERR_error_string***. Прототипы этих функций описаны в файле *openssl/err.h*.

Для зашифрования открытого текста используются две функции:

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx,
                      unsigned char *out,
                      int *outl, const unsigned char *in,
                      int inl);
```

и

```
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx,
                     unsigned char *out, int *outl);
```

Параметр *ctx* является адресом структуры контекста алгоритма. Параметры *out* и *in* являются, соответственно, выходным и входным буферами, а *outl* и *inl* их длинами (параметр *outl* в процессе работы функции может изменяться). Функции возвращают 1, если шифрование завершено успешно и 0 в противном случае.

Функция ***EVP_EncryptUpdate*** зашифровывает данные порциями, размер которых может быть больше размера блока. Обычно все же длину буфера делают кратной длине блока. Если используется режим шифрования CBC, то он требует дополнения последнего блока. По умолчанию, режим создания дополнения включен. За его включение/отключение отвечает функция:

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *c, int pad);
```

Если при ее вызове в качестве параметра *pad* задать 1, то режим создания дополнения будет включен (по умолчанию), а если 0, то выключен.

Функция ***EVP_EncryptUpdate*** в режиме включенного дополнения зашифровывает порции данных размером, кратным длине блока (в

нашем случае 16 байт). Если последний фрагмент имеет длину не кратную длине блока, то функция ***EVP_EncryptUpdate*** зашифрует максимально возможную его часть с длиной, кратной длине блока. Для шифрования оставшихся данных вызывается функция ***EVP_EncryptFinal***. Эта же функция вызывается и в том случае, если последний фрагмент имеет длину, кратную длине блока и нужно добавить еще один блок.

Если же используется режим шифрования, не требующий дополнения неполных блоков, например CFB, то режим создания дополнения отключается. В этом случае функции ***EVP_EncryptUpdate*** на вход подают фрагменты данных любого размера и на выходе получают шифртекст такого же размера. А вызов функции ***EVP_EncryptFinal*** не требуется.

При расшифровании данных, с контекстом алгоритма проводят те же действия по первоначальной инициализации, но для подготовки контекста используют функцию ***EVP_DecryptInit***. Функция имеет ту же сигнатуру, что и ***EVP_EncryptInit***. Естественно, алгоритм, режим шифрования, значения ключа и вектора инициализации должны быть теми же самыми, что и при зашифровании.

Непосредственно для расшифрования данных используются функции ***EVP_DecryptUpdate*** и ***EVP_DecryptFinal***. Сигнатуры этих функций совпадают с сигнатурами функций, использовавшихся для зашифрования данных. Есть некоторый нюанс, связанный с расшифрованием последнего блока зашифрованных данных в режиме CBC. Для корректного расшифрования данных режим дополнения с помощью функции ***EVP_CIPHER_CTX_set_padding*** должен быть отключен для всех блоков, кроме последнего. Что касается режима CFB, то также как и при зашифровании, режим дополнения отключен постоянно.

В данной работе предполагается, что симметричное шифрование в программе будет проводиться с файлом, выбранным пользователем. Для доступа к файлу можно использовать обычные функции файлового ввода-вывода библиотеки языка C/C++, функции Win32 API или рассмотренный выше тип файлового *BIO*. Что касается шифрования самого сгенерированного сеансового ключа с целью его дальнейшего обмена, то оно должно проводиться средствами асимметричных криптоалгоритмов, например RSA.

Асимметричное шифрование данных с помощью криптоалгоритма RSA

Для реализации технологии создания «цифрового конверта» (то есть зашифрования сеансового ключа алгоритмом с открытым ключом), будем использовать алгоритм RSA. Структуры данных и прототипы функций, отвечающие за генерацию ключевых пар, шифрование и т.д. хранятся в файле *openssl/rsa.h*.

Для хранения в памяти информации о ключевой паре используется структура *RSA*. Структура создается динамически с помощью функции ***RSA_new***:

```
RSA *RSA_new(void);
```

При ее вызове указателю на структуру *RSA* присваивается возвращаемое значение.

Для генерации ключевой пары можно использовать функцию ***RSA_generate_key***:

```
RSA *RSA_generate_key(int bits, unsigned long e,
                      void (*callback) (int, int, void *),
                      void *cb_arg);
```

Параметр *bits* задает размер ключа в битах. По соображениям криптостойкости его значение должно быть не менее 2048 бит. Следующий параметр задает значение открытой экспоненты. Обычно в качестве *e* задают константу *RSA_F4* (0x10001L). Параметр *callback* является указателем на функцию обратного вызова, которая должна демонстрировать ход генерации ключевой пары. Последний параметр также используется данной функцией. Если никакой демонстрации хода вычислений не требуется, то последние два параметра можно задавать как *NULL*.

После генерации ключевой пары можно осуществить ее проверку на пригодность с помощью функции ***RSA_check_key***:

```
int RSA_check_key(const RSA *);
```

Функция вернет одно из трех значений: 1, если тест пройден успешно, 0, если ключевая пара тест не прошла и -1, если произошла ошибка при выполнении теста.

Для выполнения операции зашифрования данных с помощью открытого ключа используется функция ***RSA_public_encrypt***:

```
int RSA_public_encrypt(int flen, const unsigned char *from,
                       unsigned char *to, RSA *rsa,
                       int padding);
```

Параметр *flen* является размером шифруемых данных. В нашем случае, если ключ алгоритма AES имеет размер 128 бит, то параметр *flen* должен быть равен 16. Следующий параметр определяет буфер с шифруемыми данными (т.е. с симметричным ключом AES). Параметр *to* задает адрес буфера, куда попадают зашифрованные данные. Размер этого буфера определятся предварительным вызовом функции ***RSA_size***:

```
int RSA_size(const RSA *rsa);
```

Если длина ключа алгоритма RSA (числа *n*) была задана как 2048 бит, то функция ***RSA_size*** вернет значение 256 байт. Параметр *padding* задает способ дополнения. Его рекомендуют задавать равным константе *RSA_PKCS1_OAEP_PADDING*. Длина входных данных при этом должна быть на 41 байт меньше размера выходного буфера. В нашем случае это требование выполняется с большим запасом.

Для расшифрования используется функция:

```
int RSA_private_decrypt(int flen, const unsigned char *from,
                        unsigned char *to, RSA *rsa,
                        int padding);
```

Ее параметры аналогичны по назначению параметрам функции ***RSA_public_encrypt***. Параметр *padding* должен иметь то же значение.

Освобождение памяти, занятой структурой RSA, осуществляется функцией:

```
void RSA_free(RSA *r);
```

Для того чтобы сохранить сгенерированную ключевую пару в файле и извлечь ее при необходимости, можно использовать два вида функций. Одни выгружают пару в двоичном формате *DER* (*Distinguished Encoding Rules*), другие в формате *PEM* (*Privacy Enhanced Mail*), который представляет собой DER-формат, закодированный кодировкой *base64*. Далее будут рассмотрены функции для работы с форматом PEM, прототипы которых определены в файле *openssl/pem.h*.

Открытый и закрытый ключи записываются в разные файлы. Для записи открытого ключа применяется функция:

```
int PEM_write_bio_RSAPublicKey(BIO *bp, RSA *x);
```

В качестве параметра *bp* можно использовать файловый *BIO*, открытый для записи. Функция возвращает 1 в случае удачного завершения и 0 в противном случае.

Закрытый ключ, как правило, хранится в зашифрованном виде. Он шифруется с помощью блочного шифра с ключом, получаемым из хэшированного пароля. Для выгрузки закрытого ключа может использоваться функция:

```
int PEM_write_bio_RSAPrivateKey(BIO *bp, RSA *x,
    const EVP_CIPHER *enc, unsigned char *kstr, int klen,
    pem_password_cb *cb, void *u);
```

Параметр *enc* представляет собой константный указатель на структуру *EVP_CIPHER*, вместо которого можно произвести вызов функции нужного алгоритма, например, ***EVP_aes_128_cfb128()***.

Можно указать в параметре *cb* адрес функции обратного вызова, которая будет запрашивать пароль (ее пример есть в документации). Если указатель *kstr* не будет равен *NULL*, то в качестве пароля будут использованы первые *klen* символов из массива, на который указывает *kstr*, при этом параметр *cb* игнорируется. Если *cb* равен *NULL*, а параметр *u* не равен *NULL*, то *u* интерпретируется как строка, заканчивающаяся нулем, и эта строка используется как пароль. Также можно все параметры (*kstr, cb, u*) установить в *NULL*, и библиотека запросит у пользователя ввод парольной фразы (более 3-х символов) в консоли (причем с отключенным эхо-выводом).

Для извлечения закрытого ключа из файла применяется функция:

```
RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **x,
                                pem_password_cb *cb,
                                void *u);
```

Функция в качестве параметра *x* получает адрес указателя на созданную структуру *RSA* или *NULL*, если ее необходимо создать. Остальные параметры аналогичны функции ***PEM_write_bio_RSAPrivateKey***. Функция возвращает указатель на структуру с извлеченным ключом или *NULL* в случае ошибки.

Для извлечения открытого ключа используется функция:

```
RSA *PEM_read_bio_RSAPublicKey(BIO *bp, RSA **x,
                               pem_password_cb *cb,
                               void *u);
```

Ее параметры аналогичны предыдущей функции. Причем последние два параметра всегда задаются равными *NULL*, так как при выгрузке в файл открытого ключа шифрование не производится. Функция также возвращает указатель на структуру с извлеченным ключом или *NULL* в случае ошибки.

Симметричное шифрование данных с помощью криптоалгоритма ГОСТ 28147-89

Ранее рассказывалось, как сконфигурировать пакет OpenSSL для использования отечественных симметричных и асимметричных криптоалгоритмов, реализация которых была добавлена в версии 1.0.0 компанией «Криптоком». В дальнейшем описании будем исходить из того, что конфигурационный файл дополнен информацией, позволяющей загрузить модуль *gost* и создана переменная среды *OPENSSL_CONF*, содержащая путь к этому конфигурационному файлу.

Реализация отечественных криптоалгоритмов в OpenSSL осуществляется с использованием технологии дополнительных подгружаемых модулей (*engine*). Поэтому перед началом работы с модулем, реализующим поддержку алгоритмов ГОСТ, его необходимо активировать. Для этого существует несколько вариантов. Самым простым является считывание конфигурационного файла, в котором заранее определяются параметры загрузки модуля *gost* (как было показано выше). Для этого можно использовать функцию ***OPENSSL_config***, прототип которой определен в заголовочном файле *openssl/conf.h*.

Ранее предлагался вариант инициализации OpenSSL путем вызова двух функций: ***OpenSSL_add_all_algorithms*** и ***ERR_load_error_strings***. Если предполагается считывание конфигурационного файла, путь к которому определен в переменной среды *OPENSSL_CONF*, то эту операцию можно совместить с вызовом функции ***OpenSSL_add_all_algorithms***. Дело в том, что в действительности это макрос, который в зависимости от того, определена ли символическая константа *OPENSSL_LOAD_CONF*, замещается вызовом одной из двух функций:

```
void OPENSSL_add_all_algorithms_noconf(void);
void OPENSSL_add_all_algorithms_conf(void);
```

Как правило, константа не определяется и вызывается первый вариант, который приводит только к загрузке алгоритмов. Если вместо макроса ***OpenSSL_add_all_algorithms*** вызвать непосредственно функцию ***OPENSSL_add_all_algorithms_conf***, то это позволит осуществить не только загрузку алгоритмов, но и считывание стандартного конфигурационного файла.

В целом для шифрования с помощью ГОСТ 28147-89 используются те же функции с префиксом *EVP*, что и при шифровании алгоритмом

AES. Небольшое отличие заключается в процессе настройки контекста алгоритма для выполнения зашифрования или расшифрования. Рассмотренные ранее функции ***EVP_EncryptInit*** и ***EVP_DecryptInit***, выполнявшие эти операции, данном случае использовать нельзя, так как они ориентированы на вызов стандартного модуля реализации криптоалгоритмов. Поэтому для работы с модулем *gost* будем использовать их расширенные варианты: ***EVP_EncryptInit_ex*** и ***EVP_DecryptInit_ex*** (подробнее будут рассмотрены ниже), которые содержат дополнительный параметр, определяющий модуль реализации алгоритма. Он представляет собой указатель на структуру *ENGINE*, который можно предварительно получить с помощью функции ***ENGINE_by_id***.

```
ENGINE *ENGINE_by_id(const char *id);
```

Объявления структуры и функции находятся в файле *openssl/engine.h*. Единственным параметром функции является константный указатель на строку *id*, содержащую имя модуля (*engine*). С учетом имени, заданного нами в конфигурационном файле, функцию можно вызвать так:

```
ENGINE *engine_gost = ENGINE_by_id("gost");
```

В случае ошибки, функция возвращает значение *NULL*.

Порядок действий, производимых для зашифрования или расшифрования, по сравнению с применением криптоалгоритма AES, в целом не меняется. Также в функциях используются структуры *EVP_CIPHER* и *EVP_CIPHER_CTX*. При шифровании криптоалгоритмом AES можно было использовать специализированные функции типа ***EVP_aes_128_cbc***, которые возвращали константный указатель на структуру *EVP_CIPHER*. Для алгоритмов ГОСТ таких специальных функций нет, поэтому используется более общий способ получения константного указателя на структуру *EVP_CIPHER*. В частности, его можно получить, используя заданное имя алгоритма или численный идентификатор с помощью одной из функций:

```
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
const EVP_CIPHER *EVP_get_cipherbynid(int nid);
```

В качестве параметров данных функций можно использовать символические константы *SN_id_Gost28147_89* и *NID_id_Gost28147_89* соответственно, объявленные в файле *openssl/obj_mac.h* (включать его в текст программы отдельно не нужно). Первая из констант замещается препроцессором на строку

"gost89", которую также непосредственно можно указывать в качестве параметра функции ***EVP_get_cipherbyname***.

В созданной при вызове одной из двух описанных выше функций структуре *EVP_CIPHER* содержатся параметры алгоритма, в том числе режим шифрования. По умолчанию устанавливается режим CFB (гаммирование с обратной связью). В этом можно убедиться, вызвав функцию:

```
int EVP_CIPHER_mode(const EVP_CIPHER *e);
```

Она вернет текущий режим в виде одной из предопределенных констант. Режиму CFB соответствует константа *EVP_CIPH_CFB_MODE* (со значением 3). Что касается режима CBC, то, как известно, в стандарте ГОСТ 28147-89 он для шифрования не используется и, следовательно, в данной реализации отсутствует. Программист при необходимости может разработать собственную реализацию данного режима, если это потребуется. Мы же в данной работе ограничимся использованием устанавливаемого по умолчанию режима CFB.

Со структурой типа *EVP_CIPHER_CTX* производятся абсолютно те же действия, что и при шифровании алгоритмом AES. Сначала происходит инициализация контекста шифрования с помощью функции ***EVP_CIPHER_CTX_init*** (при статическом размещении переменной) или ***EVP_CIPHER_CTX_new*** (в случае динамической переменной). Далее контекст алгоритма подготавливается к выполнению операции зашифрования с помощью вызова уже упомянутой ранее функции:

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx,
                       const EVP_CIPHER *cipher,
                       ENGINE *impl,
                       const unsigned char *key,
                       const unsigned char *iv);
```

Единственным отличием сигнатуры данной функции от функции ***EVP_EncryptInit*** является параметр *impl*, представляющий собой указатель на структуру *ENGINE*, предварительно получаемый с помощью функции ***ENGINE_by_id***. Ниже показан пример инициализации и установки параметров контекста шифрования:

```
unsigned keybuf[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
unsigned char iv[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
ENGINE *engine_gost = ENGINE_by_id("gost");
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
```

```
int ret = EVP_EncryptInit_ex(&ctx,
          EVP_get_cipherbynid(NID_id_Gost28147_89),
          engine_gost, (unsigned char*) keybuf, iv);
```

Для простоты здесь не указывается обработка возвращаемого функцией ***EVP_EncryptInit_ex***, а ключ и синхропосылка заданы непосредственно. В реальности же необходимо формировать их псевдослучайно. После вызова функции ***EVP_EncryptInit_ex*** можно осуществлять зашифрование открытого текста функцией ***EVP_EncryptUpdate*** так же, как это делалось при использовании алгоритма AES. Поскольку установлен режим шифрования CFB, то вызов функции ***EVP_EncryptFinal*** не требуется. По окончании зашифрования контекст алгоритма также освобождается вызовом функций ***EVP_CIPHER_CTX_cleanup*** или ***EVP_CIPHER_CTX_free***.

Подготовка контекста алгоритма к расшифрованию шифртекста аналогична, только производится вызов функции ***EVP_DecryptInit_ex***, имеющей сигнатуру, аналогичную функции ***EVP_EncryptInit_ex***. Расшифрование блоков шифртекста также производится последовательным вызовом функции ***EVP_DecryptUpdate***.

Среди отечественных криптоалгоритмов нет такого, который можно было бы использовать непосредственно для реализации процедуры обмена сеансовым ключом, подобно RSA. В отечественной криптографии реализованы только алгоритмы электронной подписи, которые не позволяют восстанавливать зашифрованную ранее с их помощью информацию в процессе верификации. В частности, в составе OpenSSL реализован алгоритм из стандарта ГОСТ Р 34.10-2001, основанный на использовании эллиптических кривых.

Однако, ключевые пары такого алгоритма, принадлежащие двум участникам протокола защищенного обмена данными и операции над точками эллиптической кривой, определенные стандартом, могут быть использованы для реализации алгоритма согласования ключей *VKO GOST R 34.10-2001*, описанного в RFC 4357. Он представляет собой вариант алгоритма *ECDH* (*Elliptic curve Diffie–Hellman*, алгоритм Диффи-Хелмана на эллиптических кривых). Согласованный с его помощью 256-битный ключ далее используется для шифрования сеансового ключа с помощью блочного шифра ГОСТ 28147-89. Далее будет рассмотрен порядок генерации ключевых пар алгоритма ГОСТ Р 34.10-2001, их загрузки в файл и выгрузки из файла, а также непосредственно процесс согласования ключа шифрования сеансового ключа.

Согласование ключа с помощью алгоритма VKO GOST R 34.10-2001

Для реализации процедуры согласования ключа будем использовать функции, прототипы которых описаны в заголовочном файле *openssl/evp.h*. Сначала рассмотрим процесс генерации ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001.

Первым делом необходимо создать и инициализировать контекст для операций с ключевыми парами. Для их хранения применяется структура *EVP_PKEY*. В программе объявляется переменная-указатель на эту структуру, а сама она (пока пустая без каких-либо параметров и значений ключевой пары) создается в динамической памяти с помощью функции:

```
EVP_PKEY *EVP_PKEY_new(void);
```

Если в дальнейшем потребуется освободить память, выделенную под структуру типа *EVP_PKEY*, то можно использовать функцию:

```
void EVP_PKEY_free(EVP_PKEY *pkey);
```

Также необходимо объявить указатель еще на одну структуру: *EVP_PKEY_CTX*. Она содержит непосредственно сам контекст для выполнения операций с ключевыми парами. Для создания контекста, который будет использоваться при генерации ключевой пары, необходимо использовать функцию:

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new_id(int id, ENGINE *e);
```

Параметр *e*, представляет собой указатель на структуру *ENGINE*, также предварительно получаемый с помощью функции ***ENGINE_by_id***. Параметр *id* является предопределенным числовым идентификатором (*NID*), который соответствует нужному алгоритму. Для алгоритма ГОСТ Р 34.10-2001 это *NID_id_GostR3410_2001*. Функция возвращает значение, которое нужно присвоить указателю на структуру типа *EVP_PKEY_CTX*. В случае ошибки вернется *NULL*.

Перед генерацией ключевой пары необходимо установить набор параметров алгоритма. Наборы определены в RFC 4357 и обозначаются одной или двумя латинскими буквами: A, B, C, XA, XB (можно также указывать соответствующий OID). Какой из наборов использовать в данном случае без разницы, лишь бы они совпадали в обеих ключевых парах, так как эти параметры используются при вычислении общего ключа. Установить набор можно с помощью функции:

```
int EVP_PKEY_CTX_ctrl_str(EVP_PKEY_CTX *ctx,
                          const char *type,
                          const char *value);
```

Функция передает контексту, ранее полученному с помощью вызова функции ***EVP_PKEY_CTX_new_id***, команду с названием, представляемым строковой константой (параметр *type*). Для алгоритма ГОСТ Р 34.10-2001 имеется только одна команда – *"paramset"*. В качестве параметра *value* передается строковая константа с названием набора параметров, например *"A"*.

После установки параметров алгоритма, необходимо инициализировать контекст для создания ключевой пары. Это делается с помощью вызова функции:

```
int EVP_PKEY_keygen_init(EVP_PKEY_CTX *ctx);
```

Генерация ключевой пары осуществляется вызовом функции:

```
int EVP_PKEY_keygen(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey);
```

В качестве параметров функции передается указатель на структуру с контекстом и адрес указателя (так как параметр изменяется) на структуру для размещения сгенерированной ключевой пары.

После генерации ключевой пары ее необходимо сохранить в файлах, содержащих открытый и закрытый ключ. Также как и для хранения ключей алгоритма RSA, будем использовать выгрузку в файл PEM-формата. Для сохранения в файле закрытого ключа можно использовать функцию:

```
int PEM_write_bio_PrivateKey(BIO *bp, EVP_PKEY *x,
    const EVP_CIPHER *enc, unsigned char *kstr,
    int klen, pem_password_cb *cb, void *u);
```

Параметр *x* является указателем на структуру типа *EVP_PKEY*, содержащую сгенерированную ключевую пару. Остальные параметры аналогичны по своему назначению одноименным параметрам функции ***PEM_write_bio_RSAPrivateKey***, рассмотренной ранее.

Выгрузка в файл открытого ключа осуществляется вызовом функции:

```
int PEM_write_bio_PUBKEY(BIO *bp, EVP_PKEY *x);
```

Ее параметры аналогичны одноименным у рассмотренной выше функции ***PEM_write_bio_PrivateKey***.

Загрузка закрытого ключа из файла осуществляется функцией:

```
EVP_PKEY *PEM_read_bio_PrivateKey(BIO *bp, EVP_PKEY **x,
                                  pem_password_cb *cb,
                                  void *u);
```

В качестве параметра *x* передается адрес указателя на структуру типа *EVP_PKEY*, в которую будет записан извлекаемый из файла закрытый ключ. Назначение остальных параметров аналогично функции ***PEM_read_bio_RSAPrivateKey***. Возвращаемое функцией значение присваивается указателю на структуру типа *EVP_PKEY*. В случае ошибки возвращается *NULL*.

Для загрузки открытого ключа используется функция:

```
EVP_PKEY *PEM_read_bio_PUBKEY(BIO *bp, EVP_PKEY **x,
                              pem_password_cb *cb,
                              void *u);
```

Параметры функции аналогичны рассмотренной выше функции ***PEM_read_bio_PrivateKey***. Последние два параметра всегда задаются равными *NULL*. Функция возвращает указатель на структуру типа *EVP_PKEY*, в которую будет записан извлекаемый из файла открытый ключ. В случае ошибки возвращается *NULL*.

После генерации ключевой пары и выгрузки ее в файлы контекст должен быть освобожден функцией:

```
void EVP_PKEY_CTX_free(EVP_PKEY_CTX *ctx);
```

Далее рассмотрим процесс выработки сторонами обмена информацией общего ключа, который можно будет использовать для зашифрования псевдослучайного сеансового ключа симметричным криптоалгоритмом ГОСТ 28147-89. Для этого каждой из сторон требуются следующие данные:

1. Собственный закрытый ключ.
2. Открытый ключ другой стороны.
3. 64-битная псевдослучайная величина *UKM* (*user key material*), которая генерируется одной из сторон и передается другой стороне в открытом виде.

Алгоритм может использовать не только статические ключевые пары, но и эфемерные (временные). Мы будем далее полагать, что используются статические (постоянные) ключевые пары. Для выработки общего ключа каждая сторона создает контекст на основе своего закрытого ключа с помощью функции:

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new(EVP_PKEY *pkey, ENGINE *e);
```

Параметр *pkey* указывает на структуру, содержащую закрытый ключ ключевой пары. Если он хранится в файле, то его необходимо в эту структуру предварительно загрузить. Функция возвращает указатель на созданный контекст или *NULL* в случае ошибки.

Далее созданный контекст необходимо инициализировать для выработки общего ключа с помощью функции:

```
int EVP_PKEY_derive_init(EVP_PKEY_CTX *ctx);
```

Теперь необходимо сгенерировать и установить в качестве параметра алгоритма величину *UKM*. Первоначальную генерацию данной величины осуществляет та сторона, которая генерирует сеансовый ключ, и, следовательно, первой начинает процесс согласования ключа. Произведя установку параметра в своем контексте, она передает его в дальнейшем другой стороне в открытом виде вместе с зашифрованным сеансовым ключом и зашифрованными им данными.

Для установки параметра *UKM* можно использовать функцию:

```
int EVP_PKEY_CTX_ctrl(EVP_PKEY_CTX *ctx, int keytype,
                      int optype, int cmd, int p1,
                      void *p2);
```

Функция позволяет задать команду для реализации алгоритма, на основе ключа которого создан контекст. В качестве параметров *keytype* (тип ключа) и *optype* (тип операции) будем указывать значение -1. В качестве параметра *cmd* (код команды) будем указывать предопределенную константу *EVP_PKEY_CTRL_SET_IV*. В качестве параметров команды используются величины *p1* и *p2*. Первая, в данном случае, задает размерность буфера с данными для команды в байтах, а вторая является собственно указателем на этот буфер. Ниже показан пример установки параметра *UKM*:

```
unsigned __int64  ukm;
RAND_bytes((unsigned char *)&ukm, 8);
ret = EVP_PKEY_CTX_ctrl(ctx, -1, -1, EVP_PKEY_CTRL_SET_IV,
                        8, &ukm);
```

После установки параметра *UKM* необходимо добавить к контексту выработки общего ключа открытый ключ другой стороны. Это делается с помощью функции:

```
int EVP_PKEY_derive_set_peer(EVP_PKEY_CTX *ctx,
                             EVP_PKEY *peer);
```

Параметр *peer* является указателем на структуру, содержащую загруженный из файла открытый ключ другой стороны.

Далее можно непосредственно выработать общий ключ с помощью функции:

```
int EVP_PKEY_derive(EVP_PKEY_CTX *ctx, unsigned char *key,
                    size_t *keylen);
```

Параметр *key* является указателем на буфер размера *keylen*, в который помещается сгенерированный ключ. Если параметр *key* задать равным *NULL*, то по адресу *keylen* будет передан требуемый размер буфера. Для данного алгоритма генерируется 256-битный ключ, на основе которого создается контекст алгоритма ГОСТ 28147-89 и производится шифрование сеансового ключа. Зашифрованный сеансовый ключ выгружается в файл и отправляется другой стороне. Та, в свою очередь, получив величину *UKM* и зашифрованный сеансовый ключ, может сгенерировать общий ключ, расшифровать сеансовый ключ и использовать его для расшифрования присланных данных.

СОДЕРЖАНИЕ РАБОТЫ

Задание А

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифрование/расшифрование указанного файла блочным шифром AES со 128-разрядным ключом в режиме CBC или CFB-128 на выбор пользователя. Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может также импортироваться, см. п. 4). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно).
2. Генерация ключевой пары RSA с длиной ключа не менее 2048 бит.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Зашифрование сеансового ключа шифра AES с помощью ранее сгенерированного открытого ключа RSA и сохранение его в файле, извлечение из файла и расшифрование с помощью закрытого ключа для восстановления ранее зашифрованного им файла.

Задание Б

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифрование/расшифрование указанного файла блочным шифром ГОСТ 28147-89 в режиме CFB (устанавливается по умолчанию). Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может также импортироваться, см. п. 5). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно) и использоваться при расшифровании.
2. Генерация двух ключевых пар алгоритма ГОСТ Р 34.10-2001 с заранее определенным набором параметров.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Выработку общего для двух ключевых пар ключа симметричного шифрования, используемого для обмена сеансовым ключом. При этом величина *UKM* должна вырабатываться псевдослучайно, сохраняться в файле, а затем считываться при генерации общего ключа «второй стороной».
5. Зашифрование/расшифрование сеансового ключа блочным шифром ГОСТ 28147-89 на базе общего ключа, выработанного в п. 4, сохранение его в файле, извлечение из файла. Для зашифрования сеансового ключа можно использовать тот же вектор инициализации, что и для шифрования файла с открытым текстом.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используется криптографический пакет OpenSSL?
2. В каких операционных системах можно использовать пакет OpenSSL?
3. Как установить и сконфигурировать пакет OpenSSL.
4. Как выполняется инициализация библиотеки шифрования?
5. Какие статические и динамические библиотеки пакета OpenSSL задействуются в данной работе?
6. Данные каких заголовочных файлов пакета OpenSSL используются в этой лабораторной работе?
7. Что собой представляет тип BIO? Какие его разновидности вы знаете?

8. Какими средствами в пакете OpenSSL можно осуществлять генерацию псевдослучайных чисел?
9. Какие функции и типы данных, необходимые для выполнения симметричного шифрования алгоритмом AES, вы знаете?
10. Как можно осуществлять асимметричное шифрование алгоритмом RSA средствами пакета OpenSSL?
11. Какие функции для файловой выгрузки-загрузки открытых и закрытых ключей ключевых пар алгоритма RSA вы знаете?
12. Как активировать поддержку отечественных криптоалгоритмов в пакете OpenSSL?
13. Какие функции и типы данных используются при шифровании криптоалгоритмом ГОСТ 28147-89?
14. Как в отечественной криптографии строится процесс обмена сеансовым ключом?
15. Как осуществляется генерация ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001?
16. Какие функции используются для загрузки в файл и выгрузки из файла ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001?
17. Какие параметры используются при выработке общего ключа с помощью алгоритма VKO GOST R 34.10-2001? Какие функции и типы данных используются для реализации этого алгоритма?

ЛАБОРАТОРНАЯ РАБОТА № 5

СОЗДАНИЕ КРИПТОГРАФИЧЕСКИХ СООБЩЕНИЙ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА MICROSOFT CRYPTOAPI И ЦИФРОВЫХ СЕРТИФИКАТОВ X.509

Цель работы: ознакомиться со структурой и форматами представления сертификатов открытых ключей, способами их создания и импортирования в систему, а также получить навыки в создании криптографических сообщений средствами интерфейса Microsoft CryptoAPI.

ОСНОВНЫЕ ПОНЯТИЯ

Общие сведения о сертификатах X.509

Генерация ключей с использованием случайных (псевдослучайных) чисел является стандартным подходом при использовании средств симметричной криптографии. А обмен сеансовым ключом между пользователями обычно производится средствами асимметричной криптографии. Один из пользователей может, получив свободно распространяемый открытый ключ другого пользователя, зашифровать им сгенерированный сеансовый ключ, создав так называемый «цифровой конверт». Далее зашифрованный сеансовый ключ отправляется владельцу ключевой пары и тот с помощью закрытого ключа расшифровывает его и использует в дальнейшем обмене сообщениями.

Главная проблема такого подхода состоит в том, что не обеспечивается аутентификация пользователей. То есть пользователь не может быть уверен, что присланный ему открытый ключ действительно принадлежит заявленному лицу. На этом построена атака типа «человек посередине», когда злоумышленник перехватывает пересылаемый открытый ключ легального пользователя и заменяет его своим. Это позволяет ему в дальнейшем, перехватывая пересылаемые сообщения и перезашифровывая их ключом одного из легальных пользователей, получать доступ ко всей переписке.

Для устранения проблемы взаимной аутентификации пользователей и был разработан стандарт Международного союза телекоммуникаций (ITU) X.509. Он включает в себя описание элементов так называемых *инфраструктур открытых ключей* (*Public*

Key Infrastructure, PKI), а также процедур распределения ключей. Основным элементом схемы аутентификации являются сертификаты открытых ключей, содержащие сведения о владельце ключевой пары и его открытый ключ.

Сертификаты выдаются пользователям центрами сертификации (ЦС, *Certification Authority – CA*), сведения о которых также имеются в составе сертификата. ЦС подписывает сертификат пользователя своим закрытым ключом и далее любой желающий может проверить подлинность сертификата, верифицируя электронную подпись (ЭП) центра с помощью его открытого ключа. Таким образом, пользователю для проверки подлинности сертификата другого пользователя, нужен сертификат его ЦС. И в этом случае основным становится вопрос доверия пользователя к сертификату самого ЦС.

Как правило, отдельная PKI может развертываться в рамках корпоративной сети предприятия и содержать несколько ЦС, которые чаще всего связаны в иерархическую структуру. На вершине иерархии – корневой ЦС, имеющий, как правило, самоподписанный сертификат, то есть содержащий в своем составе ЭП, созданную с помощью своего же закрытого ключа. Этот ЦС выдает сертификаты подчиненным ЦС, которые в свою очередь уже могут выдавать сертификаты конечным пользователям или сертифицировать нижестоящие центры. В общем случае, если пользователи сертифицированы разными ЦС, то процесс проверки подлинности сертификатов может привести к проверке цепочки сертификатов ЦС, образующих путь в данной иерархии к узлу, которому пользователь доверяет. Такой узел будет называться *доверенным корневым центром* и доверие к нему будет означать доверие и ко всем нижестоящим в этой иерархии ЦС.

В настоящее время применяется третья версия стандарта (X.509 v3), согласно которой в состав сертификата входит ряд полей данных:

- ***Version (номер версии)*** – указывается десятичное значение 3 и шестнадцатеричное 0x2.
- ***Serial Number (серийный номер)*** – целочисленное значение, уникальное для данного ЦС.
- ***Signature Algorithm (идентификатор алгоритма подписи)*** – поле, определяющее использованные ЦС при создании сертификата алгоритмы хэширования и цифровой подписи.
- ***Issuer (эмитент, издатель)*** – поле, содержащее отличительное имя (*Distinguished Name, DN*) центра сертификации, выдавшего сертификат. Формат записи отличительных имен определен стандартом X.500. Он состоит из набора выражений типа «*атрибут=значение*», разделенных запятой. Например,

отличительное имя ЦС может выглядеть так: `C=RU, ST=Belgorodskaya obl., L=Belgorod, O=BSTU, CN=IT_CA`. В данной записи использованы атрибуты: `C` (*Country Name*) – двухбуквенный код страны, `ST` (*State or Province Name*) – наименование области, `L` (*Locality Name*) – наименование населенного пункта, `O` (*Organization Name*) – название организации, `CN` (*Common Name*) – общепринятое имя.

- ***Validity (Not Before/ Not After) (период действия (не ранее/не позднее))*** – значения, определяющие период, на протяжении которого сертификат действителен.
- ***Subject (субъект)*** – отличительное имя субъекта (владельца ключевой пары). У самоподписанных сертификатов значения полей *Issuer* и *Subject* совпадает.
- ***Subject Public Key Info (информация об открытом ключе субъекта)*** – содержит значение открытого ключа и идентификатор алгоритма.
- ***Расширения*** (необязательные поля, определенные в версии 3) – могут содержать информацию, которую можно разделить на три категории: информация о ключах и политиках, атрибуты субъекта и органа сертификации, ограничения маршрута сертификации. Поля могут объявляться критичными и некритичными. Некритичные поля приложение, использующее сертификат, может игнорировать.
- ***Значение подписи сертификата*** – представляет собой подписанный закрытым ключом ЦС хэш-код всех полей сертификата.

Для хранения сертификатов в запоминающих устройствах и оперирования ими приложениями разных типов необходимы единообразные форматы их представления и кодирования. Для этого применяется так называемая *абстрактная синтаксическая нотация версии 1* (*Abstract Syntax Notation One, ASN.1*). ASN.1 является гибкой нотацией, позволяющей определять как простые, так и структурированные типы данных и кодировать их совокупностью байтов (октетов). Для представления содержимого сертификатов в рамках нотации ASN.1 используются отличительные правила кодирования (*Distinguished Encoding Rules, DER*), которые обеспечивают однозначный способ кодирования каждого из значений ASN.1.

Что касается форматов файлов, содержащих сертификаты, то здесь имеется несколько вариантов. Некоторые из этих форматов представления сертификатов определены в стандарте *PEM* (*Privacy*

Enhanced Mail, почта повышенной секретности) и группе стандартов *PKCS* (*Public Key Cryptography Standards*, стандарты криптографии с открытым ключом компании RSA Security, Inc.). Перечень форматов представлен ниже:

- ***DER*** подразумевает хранение в файле непосредственно двоичного содержимого сертификата в DER-кодировке. Файлы обычно имеют расширение *.cer* или *.crt* и могут содержать только один сертификат без пути сертификации (цепочки сертификатов, ведущей к доверенному корневому ЦС).
- ***PEM*** – это сертификат в формате DER, закодированный с помощью кодировки *base64*, которая позволяет представить произвольные двоичные данные в виде последовательности печатных ASCII-символов. Закодированные данные помещаются в файле между строками «-----BEGIN CERTIFICATE-----» и «-----END CERTIFICATE-----», которые используются как ограничители начала и конца сертификата. По умолчанию файлы имеют расширение *.pem*, но можно также использовать и расширения *.cer* и *.crt*. Данный формат также не позволяет задавать доверительную цепочку сертификатов. Кроме сертификатов, в этом формате хранятся закрытые ключи (предварительно зашифрованные симметричным алгоритмом с ключом из хэшированного пароля) и запросы на сертификацию (см. ниже). Назначение содержимого файла можно узнать из названий ограничителей.
- ***PKCS#7*** изначально предназначался для определения синтаксиса криптографических сообщений (*Cryptographic Message Syntax, CMS*), в которых применялись бы шифрование и/или ЭП. Однако можно создать и вырожденное сообщение, которое не имеет данных, а содержит только сами сертификаты. Кроме того, в файле этого формата можно разместить все сертификаты, входящие в доверенный путь сертификации. Файлы имеют расширение *.p7b* или *.p7c*.
- ***PKCS#12*** является единственным вариантом хранения сертификата, закрытого ключа и доверительной цепочки сертификации в одном файле. Он используется в основном для экспорта закрытого ключа. Данные шифруются симметричным алгоритмом с ключом из хэшированного пароля. Файл может иметь расширение *.p12* или *.pfx*.

Кроме вышеперечисленного, также для создания запроса пользователя на сертификацию применяется формат *PKCS#10*. В

запросе указываются данные субъекта, его открытый ключ, необходимые параметры и далее запрос подписывается закрытым ключом субъекта. Хранится запрос обычно в файле PEM-формата с соответствующими названиями ограничителей.

Стандарт X.509 помимо сертификатов определяет много и других элементов инфраструктур открытых ключей. Однако в данной работе мы не будем рассматривать работу реальной PKI, а ограничимся использованием сертификатов по сути, как контейнеров ключей, которые могут устанавливаться в операционной системе и использоваться для защищенного обмена данными. Будем считать, что созданные в процессе выполнения работы сертификаты принадлежат простейшей PKI, содержащей один ЦС с самоподписанным сертификатом.

В составе настольных версий ОС Windows имеются средства для управления сертификатами, но нет средств их создания. В данной работе для этих целей мы будем использовать криптографический пакет OpenSSL.

Создание сертификатов X.509 с помощью OpenSSL

Пакет OpenSSL позволяет развертывать PKI, что подразумевает возможность создания сертификатов X.509 и дальнейшего оперирования ими. Воспользуемся этим функционалом пакета, осуществляя взаимодействие с ним через командную строку в режиме консольного приложения. Далее вкратце опишем процесс создания сертификатов X.509.

В прошлой работе была показана процедура настройки пакета с помощью конфигурационного файла. Внесем дополнительные изменения в ряд секций этого файла с помощью любого текстового редактора. В частности, в секцию **[req_distinguished_name]**, которая содержит значения переменных, определяющих параметры отличительных имен, указываемых в запросах на сертификацию. Установим новые значения переменных, отвечающих за значения атрибутов имен по умолчанию:

```
countryName_default             = RU
stateOrProvinceName_default     = Belgorodskaya obl.
localityName_default            = Belgorod
0.organizationName_default      = My_organization
```

Содержимое секции **[req_attributes]** удалим полностью (но название секции оставим!). Секция **[policy_match]** определяет какие из элементов отличительного имени субъекта в запросе на

сертификацию должны совпадать с соответствующими элементами имени ЦС. По умолчанию совпадать должны названия страны, области и организации (параметры установлены в значение *match*). При желании их можно изменить на значения *supplied* (предоставленный) и *optional* (необязательный).

OpenSSL предоставляет возможность использования тестового ЦС для создания сертификатов открытых ключей. В конфигурационном файле в секции **[CA_default]** определены параметры по умолчанию работы пакета в режиме ЦС. В частности параметр **dir**, который определяет местоположение каталога ЦС, изначально установлен в значение *./demoCA*. Вместо этого введем непосредственный путь к каталогу, например:

```
dir = C:/OpenSSL-Win32/demoCA
```

После этого сохраним конфигурационный файл и закроем его.

Теперь по указанному пути создадим каталог с именем *demoCA*. Далее необходимо создать набор файлов и подкаталогов, требуемый для функционирования тестового ЦС. Сначала в созданном каталоге *demoCA* создадим два текстовых файла: пустой с именем *index.txt* и файл *serial* со значением серийного номера (например, *01*), который будет присвоен следующему подписанному сертификату. Также в каталоге *demoCA* создадим подкаталог *newcerts*, в который будут помещаться копии создаваемых сертификатов.

В состав OpenSSL входит большое число команд, имеющих многочисленные параметры и их подробное рассмотрение в рамках данной работы невозможно. Поэтому рассмотрим несколько необходимых команд, которые понадобятся для выполнения данной работы.

Для выполнения работы нам понадобится создать:

- ключевую пару алгоритма RSA и самоподписанный сертификат ЦС в PEM-формате, которые будут использоваться для создания сертификатов конечных пользователей;
- две ключевые пары RSA и сертификаты пользователей в PEM-формате, подписанные созданным ранее закрытым ключом ЦС;
- сертификаты пользователей в формате PKCS#12, созданные на базе ключевых пар и сертификатов из предыдущего пункта. Они понадобятся непосредственно для создания и расшифрования криптографических сообщений;

Рассмотрим процесс создания данных сертификатов командами OpenSSL на конкретных примерах, с учетом изменений, внесенных ранее в конфигурационный файл. Все приведенные далее примеры

команд предполагают, что файл *openssl.exe* уже запущен и команды вводятся после приглашения **OpenSSL**>.

Создание самоподписанного сертификата выполним с помощью команды **req**. Вообще эта команда предназначена для создания запросов на сертификацию, но если указана опция **-x509**, то создается самоподписанный сертификат. Ниже показан пример команды, которая одновременно выполняет генерацию ключевой пары RSA с длиной 2048 бит (опция **-newkey**), задает срок действия сертификата в 2 года (опция **-days**), записывает созданную ключевую пару в файл *ca_test_key.pem* текущего каталога (опция **-keyout**), а сертификат – в файл *ca_test_cert.pem* (опция **-out**):

```
req -x509 -newkey rsa:2048 -days 730 -keyout ca_test_key.pem -out ca_test_cert.pem
```

После запуска команды необходимо ввести ряд данных. Сначала это ввод и подтверждение пароля, который будет использован для шифрования закрытого ключа. Его считывание происходит из стандартного потока ввода без эхо-вывода на экран и завершается после нажатия клавиши **Enter**. Затем пользователь должен ввести атрибуты отличительного имени:

```
Country Name (2 letter code) [RU]:
State or Province Name (full name) [Belgorodskaya obl.]:
Locality Name (eg, city) [Belgorod]:
Organization Name (eg, company) [My_organization]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:CA_Test
Email Address []:
```

Значения атрибутов вводятся латинскими буквами, ввод завершается нажатием клавиши **Enter**. Слева в квадратных скобках указываются значения по умолчанию, взятые из конфигурационного файла. Не вводя каких-либо символов и нажав клавишу **Enter**, пользователь соглашается со значением по умолчанию. Если значение по умолчанию в файле не указано, то скобки остаются пустыми. Нажав клавишу **Enter** в этом случае, пользователь игнорирует такой атрибут. В приведенном примере мы согласились со значениями по умолчанию для атрибутов: *Country Name*, *State or Province Name*, *Locality Name*, *Organization Name*, проигнорировали атрибуты: *Organizational Unit Name*, *Email Address* и установили для атрибута *Common Name* значение *CA_Test*.

Созданные сертификат и закрытый ключ в файлах представлены только в кодировке *base64*. Увидеть сертификат в текстовом виде в окне консоли можно, введя команду:

```
x509 –in ca_test_cert.pem –noout –text
```

Выведя сертификат в окно консоли можно убедиться, что отличительные имена эмитента и субъекта совпадают. В разделе расширений можно увидеть, что в подразделе **X509v3 Basic Constraints** (основные ограничения) значение параметра **CA** установлено в значение **TRUE**, что означает принадлежность данного сертификата центру сертификации. У сертификата обычного пользователя этот параметр будет установлен в **FALSE.**

Для просмотра файла закрытого ключа используется аналогичная команда, но вместо **x509** нужно указать **rsa**. После ввода команды необходимо будет указать парольную фразу.

Создание ключевой пары и сертификата пользователя проводится в два этапа. Сначала командой **req** создается запрос на сертификацию с одновременной генерацией ключевой пары, а затем с помощью команды **ca** он подписывается закрытым ключом ЦС. Создание запроса:

```
req -newkey rsa:2048 -keyout User_A_key.pem -out User_A_req.pem
```

Команда создает новый закрытый ключ RSA длиной 2048 бит, помещает его в файл *User_A_key.pem*, а запрос помещает в файл *User_A_req.pem*. Далее также вводится пароль и отличительное имя. Значения всех атрибутов (кроме *Common Name*) зададим аналогичными тем, которые содержатся в сертификате ЦС. В качестве значения атрибута *Common Name* укажем *User_A.*

Теперь необходимо подписать созданный запрос на сертификацию с помощью сертификата ЦС и его закрытого ключа. Для этого будет использоваться команда **ca**, которая имитирует работу ЦС. Создание сертификата из ранее созданного запроса произведем командой:

```
ca -md sha256 -keyfile ca_test_key.pem -cert ca_test_cert.pem -in User_A_req.pem -out User_A_cert.pem
```

Данная команда определила использование для создания ЭП хэш-функции SHA-256, в качестве закрытого ключа ЦС указан файл *ca_test_key.pem*, в качестве сертификата ЦС файл *ca_test_cert.pem*, запрос на сертификацию взят из файла *User_A_req.pem*, а созданный сертификат помещается в файл *User_A_cert.pem*. После запуска команды необходимо ввести пароль для доступа к закрытому ключу

ЦС, а затем подтвердить создание ЭП и формирование сертификата на базе запроса. В выходном файле размещается сначала текстовая версия сертификата, а затем его вариант в формате PEM. Если бы был необходим сертификат без текстового варианта, то в предыдущую команду надо было добавить опцию **-notext**. Сертификат по умолчанию создается с периодом действия 1 год (параметр **default_days** секции **[CA_default]** конфигурационного файла изначально установлен в значение 365).

Теперь на базе сертификата пользователя в формате PEM и его закрытого ключа создадим сертификат в формате PKCS#12:

```
pkcs12 -export -in User_A_cert.pem -inkey User_A_key.pem
-out User_A_cert.p12
```

После запуска команды необходимо будет ввести пароль доступа к закрытому ключу в файле *User_A_key.pem* и новый пароль экспорта для дальнейшего доступа к созданному сертификату.

Аналогично создадим ключевую пару, сертификаты в форматах PEM и PKCS#12 для второго пользователя (*User_B*).

Созданные сертификаты далее будут использованы для создания и расшифрования криптографических сообщений. Но сначала их необходимо импортировать в состав ОС Windows.

Управление сертификатами в ОС Windows

Для безопасного хранения сертификатов Windows использует *хранилище сертификатов* отдельно для каждого пользователя, так чтобы другой пользователь не смог ими воспользоваться. Физическое хранилище разделено на несколько логических. Каждое такое хранилище предназначено для сертификатов определенных функций и назначений.

Для того чтобы импортировать сертификат ЦС (*ca_test_cert.pem*), изменим его расширение с *.pem* на *.crt*. После в окне Проводника дважды щелкнем по значку этого файла. В появившемся диалоговом окне отображены сведения о сертификате, а также сообщение о том, что к этому корневому сертификату нет доверия. Если нажать на кнопку *Установить сертификат*, то появится окно *Мастера импорта сертификатов*. В этом окне нажмем кнопку *Далее*, после чего появится окно следующего шага, где нужно определить в какое логическое хранилище будет помещен импортируемый сертификат. Установим зависимый переключатель в положение *Поместить все сертификаты в следующее хранилище* и нажмем кнопку *Обзор*. В

появившемся списке выберем *Доверенные корневые центры сертификации* и нажмем *OK*, затем *Далее*. В завершающем окне нажмем кнопку *Готово*. На экране появится окно с предупреждением о безопасности, в котором сказано, что не удается проверить данный сертификат и необходимо выбрать, устанавливать ли данный сертификат или нет. Нажав *Да* завершим процесс установки.

Теперь установим сертификаты в формате PKCS#12. Сначала дважды щелкнем по файлу *User_A_cert.p12*. В первом окне Мастера нажмем кнопку *Далее*. В следующем окне, где предлагается уточнить имя импортируемого файла, также нажмем *Далее*. В следующем окне необходимо ввести пароль, который был указан при создании сертификата. Также установим флажок *Пометить этот ключ как экспортируемый*, так как в дальнейшем нам может понадобится экспортировать ключевую пару. Далее выберем в качестве хранилища *Личное* и завершим процесс импорта. Аналогично установим сертификат из файла *User_B_cert.p12*.

Для управления установленными сертификатами в Windows имеется оснастка *Сертификаты*. Отобразить ее можно запустив на выполнение файл *certmgr.msc*. В ее окне слева расположен перечень логических хранилищ. Для того чтобы просмотреть содержимое хранилища *Личное* раскроем его структуру и щелкнем по значку *Сертификаты*. Справа будут отображены установленные сертификаты с именами *User_A* и *User_B*, которые мы только что установили. Можно увидеть имя выдавшего их ЦС и срок действия. Используя меню *Действия* можно открыть сертификат, просмотреть его свойства, копировать, удалить и т.д. Если раскрыть хранилище *Доверенные корневые центры сертификации*, то можно просматривая его содержимое увидеть и установленный нами сертификат с именем *CA_Test*.

Создание криптографических сообщений с помощью функций интерфейса CryptoAPI 2.0

В предыдущей работе процесс защищенного обмена сообщениями предполагал создание отдельных файлов с зашифрованным сообщением, зашифрованным сеансовым ключом и открытым ключом, которые могли пересылаться между двумя пользователями. При этом форматы пересылаемых данных должны были заранее согласовываться и для разных групп пользователей они могли быть различными.

Для стандартизации таких форматов *RSA Laboratories* предложила спецификацию *PKCS#7* (RFC 2315), приемником которой стал выпущенный *IETF* стандарт «*Cryptographic Message Syntax (CMS)*» (RFC 5652). Стандарт CMS определяет структуру криптографических сообщений, которые могут содержать в себе зашифрованные и/или подписанные данные вместе со всей необходимой для дальнейшего расшифрования и/или проверки ЭП информацией. Например, при создании зашифрованного сообщения, в его состав включается зашифрованный открытым ключом получателя сообщения сеансовый ключ. В качестве источников ключевой информации асимметричных алгоритмов при создании криптографических сообщений используются установленные в операционной системе сертификаты. Мы будем использовать ранее сформированные и установленные в системе сертификаты пользователей *User_A* и *User_B*, а для проверки их действительности сертификат центра сертификации *CA_Test*.

В данной работе необходимо реализовать приложение, создающее и расшифровывающее криптографическое сообщение, содержащее электронную подпись сообщения и результат его зашифрования симметричным алгоритмом. Процесс создания такого сообщения можно представить следующим обобщенным алгоритмом:

1. Открыть системные хранилища сертификатов «Личное» (*MY*) и «Доверенные корневые центры сертификации» (*ROOT*). В первое мы установили сертификаты пользователей, которые будут участвовать в «обмене» сообщениями, а второе содержит сертификат ЦС. Для простоты в нашем случае сертификат получателя сообщения содержит его полную ключевую пару, хотя в реальной ситуации, конечно же, отправитель будет располагать только сертификатом с открытым ключом получателя.
2. Сформировать и предоставить пользователю список имен владельцев сертификатов, установленных в хранилище «Личное».
3. Получить выбранные пользователем из списка имена отправителя и получателя криптографического сообщения.
4. Получить контексты сертификатов отправителя и получателя сообщения и проверить их целостность с помощью сертификата ЦС.
5. Считать из файла исходное сообщение и поместить его в буфер, созданный в динамической памяти.
6. Инициализировать параметры функции, используемой для создания криптографического сообщения и получить объем

буфера в динамической памяти, необходимого для хранения блоба с сообщением.

7. Создать в динамической памяти буфер требуемого размера и вызвать функцию создания криптографического сообщения, используя в качестве одного из параметров указатель на созданный выходной буфер.
8. Сохранить созданный блоб в указанном пользователем файле.

Для расшифрования криптографического сообщения нужно выполнить следующие действия:

1. Открыть системное хранилище, которое содержит сертификат получателя (с ключевой парой) и сертификат отправителя (с открытым ключом). В нашем случае это хранилище «Личное». Сертификат получателя нужен для того, чтобы расшифровать с помощью закрытого ключа зашифрованный сеансовый ключ, а сертификат отправителя с его открытым ключом, чтобы проверить ЭП из состава криптографического сообщения. Для простоты мы не устанавливали в системе отдельный сертификат отправителя с открытым ключом, поэтому при расшифровании будет задействован тот же сертификат в формате *PKCS#12*, который участвовал в создании сообщения. В состав сообщения помещаются данные о владельцах сертификатов, которые при расшифровании ищутся в указанном хранилище.
2. Считать из файла блоб с криптографическим сообщением и поместить его в буфер, созданный в динамической памяти.
3. Инициализировать параметры функции, используемой для расшифрования криптографического сообщения и получить объем буфера в динамической памяти, необходимого для хранения блоба с расшифрованным сообщением.
4. Создать в динамической памяти буфер требуемого размера и вызвать функцию расшифрования криптографического сообщения, используя в качестве одного из параметров указатель на созданный выходной буфер.
5. Проверить результат расшифрования и верификации ЭП. В случае успеха сохранить расшифрованное сообщение в указанном пользователем файле.

Детали этих алгоритмов будут поясняться по мере рассмотрения функций интерфейса CryptoAPI, которые используются для их реализации. Большинство из них относится к CryptoAPI версии 2.0, поэтому в создаваемом приложении необходимо подключить библиотеку *crypt32.dll*. В среде Visual Studio для этого можно

использовать библиотеку импорта *crypt32.lib*, указав этот файл в свойствах проекта или включив в текст программы директиву:

```
#pragma comment(lib, "crypt32.lib")
```

Что касается заголовочных файлов, то в текст программы необходимо включить файлы *windows.h* и *wincrypt.h*.

Все рассмотренные далее функции можно разделить на три типа:

1. Функции управления хранилищами сертификатов: ***CertOpenSystemStore***, ***CertCloseStore***.
2. Функции для работы с сертификатами: ***CertEnumCertificatesInStore***, ***CertGetNameString***, ***CertFindCertificateInStore***, ***CertGetIssuerCertificateFromStore***, ***CertFreeCertificateContext***.
3. Функции поддержки криптографических сообщений. В CryptoAPI существуют два вида таких функций: базовые и упрощенные. Необходимость использования базовых возникает достаточно редко, поэтому мы рассмотрим только упрощенные функции: ***CryptSignAndEncryptMessage***, ***CryptDecryptAndVerifyMessageSignature***.

Более подробная информация о рассмотренных функциях содержится в разделе MSDN, посвященном использованию CryptoAPI (https://msdn.microsoft.com/en-us/library/windows/desktop/aa380256%28v= vs.85%29.aspx).

CertOpenSystemStore

Функция открывает системное хранилище сертификатов и имеет следующий прототип:

```
HCERTSTORE WINAPI CertOpenSystemStore(
                  HCRYPTPROV_LEGACY hprov,
                  LPTCSTR szSubsystemProtocol );
```

Параметр *hprov* не используется и должен быть равен *NULL*. Строка *szSubsystemProtocol* содержит имя системного хранилища сертификатов. В данной работе в качестве имени будем использовать строку *“MY”* для открытия хранилища «Личное» и строку *“ROOT”* для открытия хранилища «Доверенные корневые центры сертификации».

В случае успешного завершения функция возвращает дескриптор открытого хранилища. В противном случае функция возвращает *NULL*. Открытое хранилище должно быть позднее закрыто функцией ***CertCloseStore*** (см. ниже). Пример вызова функции для открытия хранилища «Личное»:

```
HCERTSTORE hStoreMy = NULL;
if (!( hStoreMy = CertOpenSystemStore(
                  NULL,TEXT("MY"))))
{
    //Вывод сообщения об ошибке
}
```

CertCloseStore

Функция закрывает открытое ранее хранилище сертификатов и имеет следующий прототип:

```
BOOL WINAPI CertCloseStore(
             HCERTSTORE hCertStore,
             DWORD dwFlags );
```

Параметр *hCertStore* задает дескриптор закрываемого хранилища. Параметр *dwFlags* будем задавать равным нулю. Функция возвращает *TRUE* в случае успеха и *FALSE* в случае неудачи.

CertEnumCertificatesInStore

Функция используется для перечисления всех контекстов сертификатов, которые хранятся в указанном хранилище. Имеет прототип:

```
PCCERT_CONTEXT WINAPI CertEnumCertificatesInStore(
                      HCERTSTORE hCertStore,
                      PCCERT_CONTEXT pPrevCertContext );
```

Параметр *hCertStore* задает дескриптор нужного хранилища, а *pPrevCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст предыдущего сертификата, найденного в данном хранилище. Функция возвращает указатель на контекст очередного сертификата, извлеченного из хранилища или *NULL*, после извлечения всех сертификатов (или если хранилище изначально пустое). Ниже показан пример организации перечисления сертификатов в

```
PCCERT_CONTEXT pCert = NULL;
while (pCert = CertEnumCertificatesInStore(hStoreMy,
pCert))
{
    //Обработка извлеченного контекста сертификата
}
```

CertGetNameString

Функция извлекает из переданного контекста сертификата и конвертирует в строку с нулем в конце имя владельца или издателя (ЦС). В зависимости от заданных параметров, функция в качестве результата своей работы может возвращать различные части отличительного имени или какой-либо вариант альтернативного имени, заданного в соответствующем поле расширения 3-й версии стандарта X.509. Данную функцию можно использовать, например, для получения имени субъекта сертификата, извлеченного из хранилища предыдущей функцией. Имеет прототип:

```
DWORD WINAPI CertGetNameString(
             PCCERT_CONTEXT pCertContext,
             DWORD dwType,
             DWORD dwFlags,
             void *pvTypePara,
             LPTSTR pszNameString,
             DWORD cchNameString );
```

Параметр *pCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст обрабатываемого сертификата. Параметр *dwType* определяет, какую часть отличительного имени (или альтернативного) будет возвращать функция. Для получения общепринятого имени (*CN*) будем использовать в качестве этого параметра константу *CERT_NAME_SIMPLE_DISPLAY_TYPE*. Для получения имени субъекта параметр *dwFlags* будем задавать равным нулю. Также равным *NULL* будем задавать параметр *pvTypePara*. Параметр *pszNameString* является адресом буфера, принимающего возвращаемую функцией строку. Количество символов, которое может вместить данный буфер передается через параметр *cchNameString*.

Функция возвращает количество символов, переданных в буфер *pszNameString*, исключая нуль-символ. Если требуемая часть имени в сертификате отсутствует, то функция вернет единицу и по адресу *pszNameString* будет пустая строка. Пример вызова функции:

```
TCHAR szNameString[128];
if (CertGetNameString(
    pCert, CERT_NAME_SIMPLE_DISPLAY_TYPE, 0, NULL,
    szNameString, 128) > 1)
{
    //Обработка строки szNameString
}
```

CertFindCertificateInStore

Функция ищет в указанном хранилище первый (или следующий) сертификат, параметры которого совпадают с заданными критериями поиска. Например, эту функцию можно использовать, если нужно найти и извлечь из хранилища сертификат (или несколько сертификатов) по заданному общепринятому имени субъекта (*CN*). Прототип функции имеет вид:

```
PCCERT_CONTEXT WINAPI CertFindCertificateInStore(
                        HCERTSTORE hCertStore,
                        DWORD dwCertEncodingType,
                        DWORD dwFindFlags,
                        DWORD dwFindType,
                        const void *pvFindPara,
                        PCCERT_CONTEXT pPrevCertContext );
```

Параметр *hCertStore* является дескриптором открытого хранилища сертификатов. В качестве параметра *dwCertEncodingType* указывается результат побитового *ИЛИ* между константами *X509_ASN_ENCODING* и *PKCS_7_ASN_ENCODING*. Параметр *dwFindFlags* задается равным нулю. Параметр *dwFindType* может содержать одну из предопределенных констант, определяющих тип поиска. Мы будем задавать этот параметр равным константе *CERT_FIND_SUBJECT_STR*, которая определяет, что сертификат ищется по имени (атрибуту *CN* отличительного имени субъекта). В этом случае параметр *pvFindPara* является указателем на строку с именем искомого сертификата. Параметр *pPrevCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст предыдущего сертификата, найденного с этим критерием поиска. Если, как в нашем случае, сертификат с заданным именем будет существовать в единственном экземпляре, этот параметр можно задавать равным *NULL*.

Функция возвращает указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст найденного сертификата или *NULL*. Пример вызова:

```
#define MY_ENCODING_TYPE (PKCS_7_ASN_ENCODING |\
                          X509_ASN_ENCODING)
TCHAR szCertNameA[] = TEXT("User_A");
PCCERT_CONTEXT pCertA = NULL;
if(!(pCertA = CertFindCertificateInStore(
              hStoreMy, MY_ENCODING_TYPE,
              0, CERT_FIND_SUBJECT_STR,
              szCertNameA,
              NULL)))
```

```
{
    //Вывод сообщения об ошибке
}
```

CertGetIssuerCertificateFromStore

Функция позволяет извлечь из заданного хранилища контекст сертификата ЦС, указанного в пользовательском сертификате в качестве издателя. Также функция позволяет произвести простейшую проверку целостности пользовательского сертификата, используя информацию сертификата ЦС. Имеет следующий прототип:

```
PCCERT_CONTEXT WINAPI CertGetIssuerCertificateFromStore(
                      HCERTSTORE hCertStore,
                      PCCERT_CONTEXT pSubjectContext,
                      PCCERT_CONTEXT pPrevIssuerContext,
                      DWORD *pdwFlags );
```

Параметр *hCertStore* является дескриптором открытого хранилища, в котором ищется сертификат ЦС. Обычно это «Доверенные корневые центры сертификации» (*ROOT*). Параметр *pSubjectContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст пользовательского сертификата, для которого отыскивается сертификат издателя. Параметр *pPrevIssuerContext* указывает на контекст предыдущего найденного сертификата этого же ЦС (если их несколько). Если ищется первый (или единственный) сертификат, то этот параметр задается как *NULL*. Параметр *pdwFlags* позволяет задать характер осуществляемой проверки пользовательского сертификата. В качестве простейшего варианта можно задать верификацию ЭП в пользовательском сертификате открытым ключом центра сертификации, а также проверку соответствия текущего времени периоду действия сертификата. Для этого данный параметр можно представить комбинацией флагов *CERT_STORE_SIGNATURE_FLAG* и *CERT_STORE_TIME_VALIDITY_FLAG*, объединенных с помощью побитового *ИЛИ*.

Функция возвращает контекст найденного сертификата или *NULL*, в случае его отсутствия в указанном хранилище. Если сертификат ЦС найден, то нулевое значение параметра *pdwFlags* после вызова функции свидетельствует об успешности проверки пользовательского сертификата. Подробную информацию о возвращаемых ненулевых значениях параметра можно получить в соответствующем разделе MSDN.

CertFreeCertificateContext

Функция освобождает контекст ранее найденного в хранилище сертификата. Имеет следующий прототип:

```
BOOL WINAPI CertFreeCertificateContext(
              PCCERT_CONTEXT pCertContext );
```

Единственный параметр представляет собой указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст сертификата. Функция всегда возвращает ненулевое значение.

CryptSignAndEncryptMessage

Одна из упрощенных функций для создания криптографического сообщения в соответствии со стандартом CMS. Результатом работы функции будет блоб, в котором будет находиться созданное сообщение. Для его создания требуются контексты сертификата отправителя с закрытым ключом и сертификата получателя с открытым ключом (или нескольких сертификатов). В процессе работы функции исходное сообщение хэшируется, подписывается закрытым ключом отправителя, зашифровывается сгенерированным сеансовым ключом. Зашифрованное сообщение также хэшируется, подписывается, после чего ко всей этой информации добавляется зашифрованный открытым ключом получателя сеансовый ключ и создается общий упакованный набор данных. Затем эти упакованные данные вновь хэшируются и подписываются и вместе с электронной подписью составляют результат работы функции – выходной блоб. Функция имеет прототип:

```
BOOL WINAPI CryptSignAndEncryptMessage(
            PCRYPT_SIGN_MESSAGE_PARA  pSignPara,
            PCRYPT_ENCRYPT_MESSAGE_PARA  pEncryptPara,
            DWORD  cRecipientCert,
            PCCERT_CONTEXT  rgpRecipientCert[],
            const BYTE  *pbToBeSignedAndEncrypted,
            DWORD  cbToBeSignedAndEncrypted,
            BYTE  *pbSignedAndEncryptedBlob,
            DWORD  *pcbSignedAndEncryptedBlob );
```

Параметр *pSignPara* является указателем на структуру *CRYPT_SIGN_MESSAGE_PARA*, которая содержит параметры электронной подписи. В MSDN имеется подробное описание данной структуры, мы же приведем пример того, как можно инициализировать данную структуру перед вызовом функции:

```
CRYPT_SIGN_MESSAGE_PARA SignPara = {
                        sizeof(CRYPT_SIGN_MESSAGE_PARA) };
SignPara.dwMsgEncodingType = MY_ENCODING_TYPE;
SignPara.pSigningCert = pCertA;
SignPara.HashAlgorithm.pszObjId = szOID_RSA_SHA256RSA;
SignPara.cMsgCert = 1;
SignPara.rgpMsgCert = &pCertA;
```

Поле *cbSize*, определяющее размер структуры в байтах, в данном примере задается при начальной инициализации переменной. Поле *dwMsgEncodingType* как обычно является результатом побитового *ИЛИ* между константами *X509_ASN_ENCODING* и *PKCS_7_ASN_ENCODING*. Поле *pSigningCert* содержит указатель на контекст сертификата отправителя сообщения. Поле *HashAlgorithm* является в свою очередь структурой типа *CRYPT_ALGORITHM_IDENTIFIER*, поле которой *pszObjId* позволяет задать строку с OID комбинации алгоритма подписи, хэширования и шифрования (или предопределенную константу, например *szOID_RSA_SHA256RSA*).

Если сообщение необходимо подписать нескольким пользователям, то массив указателей на контексты их сертификатов передается через поле *rgpMsgCert*. Поле *cMsgCert* задает размерность этого массива. В нашем случае эти поля заданы таким образом потому, что сертификат отправителя единственный. Остальные поля структуры остаются нулевыми.

Следующий параметр функции *pEncryptPara* является указателем на структуру типа *CRYPT_ENCRYPT_MESSAGE_PARA*, которая содержит параметры шифрования сообщения. Ниже показан пример инициализации данной структуры перед вызовом функции:

```
CRYPT_ENCRYPT_MESSAGE_PARA EncryptPara = {
                        sizeof(CRYPT_ENCRYPT_MESSAGE_PARA)
                        };
EncryptPara.dwMsgEncodingType = MY_ENCODING_TYPE;
EncryptPara.ContentEncryptionAlgorithm.pszObjId =
                                szOID_NIST_AES128_CBC;
```

Назначение поля *dwMsgEncodingType* аналогично полю в предыдущей структуре. Поле *pszObjId* вложенной структуры *ContentEncryptionAlgorithm* позволяет задать OID алгоритма шифрования данных (в данном примере AES-128 в режиме CBC).

Следующая пара параметров функции: *cRecipientCert* и *rgpRecipientCert* по назначению аналогичны полям *cMsgCert* и

rgpMsgCert структуры *CRYPT_SIGN_MESSAGE_PARA*, только задают массив указателей на контексты сертификатов получателей, чьими открытыми ключами должны шифроваться сеансовые ключи. В нашем случае сертификат получателя будет единственным, также как и отправителя.

Параметр *pbToBeSignedAndEncrypted* содержит адрес буфера с исходным сообщением, а параметр *cbToBeSignedAndEncrypted* задает его размер. Результирующий блоб записывается в буфер, адрес которого задается через параметр *pbSignedAndEncryptedBlob*, а размер блоба возвращается через параметр *pcbSignedAndEncryptedBlob*. Если при вызове функции в качестве параметра *pbSignedAndEncryptedBlob* задать *NULL*, то через параметр *pcbSignedAndEncryptedBlob* вернется требуемый размер буфера. Далее буфер такого размера можно создать в динамической памяти и его адрес передать в качестве параметра *pbSignedAndEncryptedBlob* при повторном (уже результативном) вызове функции.

Функция возвращает значение *TRUE* при успешном завершении и *FALSE* при неудаче. В последнем случае код ошибки можно определить с помощью вызова функции ***GetLastError***.

CryptDecryptAndVerifyMessageSignature

Функция расшифровывает криптографическое сообщение, созданное предыдущей функцией и верифицирует входящую в его состав электронную подпись. Необходимые для этого сертификаты функция самостоятельно извлекает из указанных хранилищ. Имеет прототип:

```
BOOL WINAPI CryptDecryptAndVerifyMessageSignature(
            PCRYPT_DECRYPT_MESSAGE_PARA  pDecryptPara,
            PCRYPT_VERIFY_MESSAGE_PARA  pVerifyPara,
            DWORD  dwSignerIndex,
            const BYTE  *pbEncryptedBlob,
            DWORD  cbEncryptedBlob,
            BYTE  *pbDecrypted,
            DWORD  *pcbDecrypted,
            PCCERT_CONTEXT  *ppXchgCert,
            PCCERT_CONTEXT  *ppSignerCert );
```

Параметр *pDecryptPara* указывает на структуру типа *CRYPT_DECRYPT_MESSAGE_PARA*, которая несет информацию, необходимую для расшифрования сообщения. Приведем пример инициализации такой структуры перед вызовом функции:

```
CRYPT_DECRYPT_MESSAGE_PARA DecryptPara = {
                        sizeof(CRYPT_DECRYPT_MESSAGE_PARA)
                        };
DecryptPara.dwMsgAndCertEncodingType = MY_ENCODING_TYPE;
DecryptPara.cCertStore = 1;
DecryptPara.rghCertStore = &hStoreMy;
```

Поле *dwMsgAndCertEncodingType* инициализируется той же константой, что и аналогичные поля в рассмотренных ранее структурах. Поле *rghCertStore* задает массив дескрипторов хранилищ, открытых ранее функцией ***CertOpenSystemStore***. В нашем случае используется одно хранилище (*MY*), поэтому поле инициализируется адресом его дескриптора, а поле *cCertStore* получает значение 1.

Параметр *pVerifyPara* является указателем, содержащим адрес структуры типа *CRYPT_VERIFY_MESSAGE_PARA* с информацией, необходимой для верификации ЭП в криптографическом сообщении. Пример инициализации такой структуры:

```
CRYPT_VERIFY_MESSAGE_PARA VerifyPara = {
                       sizeof(CRYPT_VERIFY_MESSAGE_PARA) };
VerifyPara.dwMsgAndCertEncodingType = MY_ENCODING_TYPE;
```

В данном случае достаточно инициализировать только лишь поля *cbSize* и *dwMsgAndCertEncodingType*, назначение которых аналогично одноименным полям в предыдущей структуре.

Параметр *dwSignerIndex* используется в случае, если сообщение подписывало несколько пользователей и в нашем случае должен быть равен нулю. Параметр *pbEncryptedBlob* содержит адрес буфера с загруженным блобом криптографического сообщения, а параметр *cbEncryptedBlob* задает его размер. Параметр *pbDecrypted* задает адрес буфера, в который функция запишет расшифрованное сообщение, а параметр *pcbDecrypted* определяет его размер. Если при вызове функции параметр *pbDecrypted* задать равным *NULL*, то через параметр *pcbDecrypted* вернется требуемый размер буфера.

Параметры *ppXchgCert* и *ppSignerCert* после вызова функции должны указывать на контексты сертификатов, содержащих соответственно, закрытый ключ получателя сообщения и открытый ключ отправителя (в нашем случае для простоты это будет тот же сертификат с закрытым ключом, который использовался для создания сообщения). В дальнейшем эти контексты нужно будет освободить с помощью функции ***CertFreeCertificateContext***. Если же вызов функции закончится неудачей вследствие невозможности расшифровать сообщение, то эти параметры будут установлены в значение *NULL*.

Функция возвращает значение *TRUE* при успешном завершении и *FALSE* при неудаче. В последнем случае код ошибки можно определить с помощью вызова функции ***GetLastError***.

СОДЕРЖАНИЕ РАБОТЫ

1. С помощью криптографического пакета OpenSSL создать:
   - ключевую пару алгоритма RSA с длиной ключа 2048 бит и соответствующий ей самоподписанный сертификат центра сертификации;
   - две ключевые пары алгоритма RSA с длиной ключа 2048 бит и соответствующие им сертификаты в формате PKCS#12 для двух пользователей – участников процесса обмена криптографическими сообщениями. Сертификаты должны быть подписаны закрытым ключом центра сертификации.
2. Установить в системе созданные сертификаты. В отчет внести последовательность команд OpenSSL, использованных для создания сертификатов центра сертификации и пользователей.
3. Разработать на языке программирования C/C++ с использованием средств криптографического интерфейса Microsoft CryptoAPI консольное или оконное приложение, выполняющее создание криптографического сообщения по стандарту CMS из указанного пользователем файла и дальнейшего его расшифрования. Криптографическое сообщение должно содержать данные, зашифрованные алгоритмом AES-128 в режиме CBC и электронную подпись, созданную с помощью алгоритма RSA. Приложение должно предлагать пользователю перечень имен субъектов сертификатов, установленных в хранилище «Личное», и принимать его выбор имен отправителя и получателя криптографического сообщения. Перед созданием сообщения необходимо верифицировать ЭП в составе выбранных сертификатов и проверить соответствие текущей даты периоду, заданному в их составе. Созданное криптографическое сообщение необходимо выгружать в указанный пользователем файл и загружать из него в память для расшифрования. Расшифрованное сообщение также необходимо выгружать в файл, указанный пользователем.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используются сертификаты открытых ключей X.509?
2. Что такое инфраструктура открытых ключей (PKI)? Какие варианты архитектуры PKI вы знаете?
3. Какова структура сертификата X.509?
4. Как сертификаты X.509 хранятся в запоминающих устройствах? Какие форматы сертификатов вы знаете?
5. Что такое поля расширений в составе сертификата X.509?
6. Как OpenSSL настраивается для работы тестового центра сертификации?
7. Какие команды OpenSSL используются для создания сертификатов?
8. Как установить созданный сертификат в системе?
9. Какие в ОС Windows имеются средства для управления установленными сертификатами?
10. Что определяют спецификация PKCS#7 и стандарт CMS?
11. Какие функции Microsoft CryptoAPI для управления хранилищами сертификатов вы знаете?
12. Какие функции Microsoft CryptoAPI для работы с сертификатами вы знаете?
13. Как определить имена всех сертификатов в хранилище?
14. Как верифицировать сертификат?
15. Какие функции Microsoft CryptoAPI поддержки криптографических сообщений вы знаете?
16. Какие структуры данных подготавливаются перед вызовом функции, создающей криптографическое сообщение?
17. Какие структуры данных подготавливаются перед вызовом функции, расшифровывающей криптографическое сообщение?

ЛАБОРАТОРНАЯ РАБОТА № 6

РЕАЛИЗАЦИЯ ЗАЩИЩЕННОЙ ПЕРЕДАЧИ ДАННЫХ ПО ПРОТОКОЛУ TLS СРЕДСТВАМИ КРИПТОГРАФИЧЕСКОГО ПАКЕТА OPENSSL

Цель работы: ознакомиться со способами организации защищенных *TLS*-соединений средствами криптографического пакета OpenSSL, а также получить навыки в создании простейших клиентских и серверных приложений в среде Visual Studio.

ОСНОВНЫЕ ПОНЯТИЯ

Создание сертификатов открытых ключей алгоритма электронной подписи ГОСТ Р 34.10-2001

В прошлой лабораторной работе был рассмотрен процесс создания с помощью OpenSSL сертификатов X.509, содержащих ключи асимметричного алгоритма RSA, которые использовались как для обмена сеансовыми ключами, так и для создания и верификации электронной подписи (ЭП). То, что для этих целей не использовались отечественные криптоалгоритмы, было обусловлено тем, что в составе Windows по умолчанию отсутствуют криптопровайдеры с их поддержкой. В данной же работе мы будем применять только средства пакета OpenSSL, поэтому для создания защищенных соединений будем использовать сертификаты ключей отечественных криптоалгоритмов, и прежде всего алгоритма создания и проверки электронной подписи ГОСТ Р 34.10-2001.

Протокол *TLS* (Transport Layer Security), как и его предшественник *SSL* (Secure Sockets Layer) предполагают установление защищенного соединения с обязательной аутентификацией сервера и опциональной аутентификацией клиента. В данной работе будет рассмотрен простейший (и, в общем-то, самый распространенный) вариант с односторонней аутентификацией сервера. Причем под сервером в данном случае мы понимаем компьютер (узел в сети), к которому клиент хочет подключиться и передать какую-либо конфиденциальную информацию. Для аутентификации и последующего согласования сеансового ключа серверу необходимы собственные сертификат и закрытый ключ, а клиенту – сертификат (или доверительная цепочка сертификатов) центра сертификации,

который сертифицировал сервер. Если клиент доверяет сертификату центра сертификации, то он может проверить валидность сертификата сервера. После этой проверки клиент может быть уверен, что обменивается данными именно с тем узлом, за который себя выдает сервер (в реальности если сервер является веб-узлом, то атрибут *CN* его сертификата задается равным доменному имени).

Для реализации передачи данных по протоколу TLS сгенерируем самоподписанный сертификат центра сертификации и с его помощью создадим сертификат сервера. Будем предполагать, что пакет OpenSSL сконфигурирован так, как описывалось в предыдущих лабораторных работах. Запустим файл *openssl.exe* и для создания самоподписанного сертификата выполним команду:

```
req -x509 -newkey gost2001 -pkeyopt paramset:A -keyout ca_gost_key.pem -out ca_gost_cert.pem -days 730
```

Данная команда создаст новую ключевую пару алгоритма ГОСТ Р 34.10-2001 с набором параметров *"A"* (с сохранением в файле *ca_gost_key.pem*) и сертификат со сроком действия в 2 года. После запуска команды необходимо ввести пароль для шифрования закрытого ключа и данные для сертификата. Большинство параметров отличительного имени принимаем заданными по умолчанию, а атрибут *CN* определяем как *CA_GOST*. Теперь создадим запрос на получение сертификата сервера также с генерацией ключевой пары:

```
req -newkey gost2001 -pkeyopt paramset:A -keyout serv_key.pem -out serv_req.pem
```

При создании запроса также надо будет ввести пароль и данные субъекта. В качестве атрибута *CN* отличительного имени субъекта укажем строку *Server_TLS*. Теперь на основании созданного запроса нужно выпустить сертификат:

```
ca -in serv_req.pem -out serv_cert.pem -keyfile ca_gost_key.pem -cert ca_gost_cert.pem
```

Созданный сертификат и его закрытый ключ далее будут использованы для организации TLS-соединения между сервером и клиентом.

Настройка проекта Visual Studio и инициализация библиотеки OpenSSL для создания TLS-соединений

Для начала необходимо определиться с тем, какие статические и динамические библиотеки нам понадобятся для создания клиентского

и серверного приложений в среде Visual Studio. Из стандартных библиотек Windows нам дополнительно понадобится файл *ws2_32.dll*, который содержит реализацию *WinSock API* – интерфейса для созданий сетевых приложений с обменом данными через сокеты. Соответственно, будем подключать статическую библиотеку импорта *ws2_32.lib*, указывая ее имя в свойствах проекта Visual C++ или в директиве:

```
#pragma comment(lib, "ws2_32.lib")
```

Аналогично подключим статические библиотеки OpenSSL *libeay32.lib* и *ssleay32.lib*. Соответствующие им динамические библиотеки при запуске проекта на выполнение должны находиться в каталоге решения или системном каталоге, содержащем *dll*-файлы установленных приложений. Поскольку наши приложения (и клиентское и серверное) в процессе создания TLS-соединения будут использовать отечественные криптоалгоритмы, то нам также понадобится файл *gost.dll*, который должен находиться в каталоге, определенном в секции **[gost_section]** конфигурационного файла *openssl.cfg* (см. лабораторную работу №4).

Для получения прототипов нужных нам функций достаточно включить в текст программы заголовочные файлы: *WinSock2.h*, *openssl/err.h*, *openssl/ssl.h*.

Также нужно определиться с тем, как в дальнейшем для проверки работоспособности запустить скомпилированное клиентское или серверное приложения на компьютере, не содержащем среды разработки и установленного пакета OpenSSL. Для этого необходимо, чтобы в одной папке с файлом приложения (или в системном каталоге) находились все используемые динамические библиотеки. Также необходимо скопировать (можно в эту папку или в другую, по желанию) конфигурационный файл *openssl.cfg*. В самом файле в секции **[gost_section]** нужно прописать реальный путь (!) к *gost.dll*. Далее необходимо создать переменную среды *OPENSSL_CONF*, содержащую путь к этому конфигурационному файлу. После выполнения этих действий, приложение, выполняя при запуске начальную инициализацию библиотеки OpenSSL, сможет загрузить модуль *gost*. Также нужно отметить, что при запуске клиентского или серверного приложения, возможно, придется **временно отключать** брандмауэр Windows или антивирусной программы, так как он может блокировать передачу данных.

Для инициализации библиотеки традиционно будем осуществлять вызов функции ***OPENSSL_add_all_algorithms_conf*** (см. лабораторную

работу №4). Причем вызвать эту функцию нужно до вызова основной инициализирующей функции (см. ниже), чтобы шифрсьюты с поддержкой алгоритмов шифрования и хэширования ГОСТ попали в число доступных при установлении соединения.

Следующим шагом при инициализации библиотеки является вызов функции ***SSL_library_init***, имеющей следующий прототип (находится в файле *openssl/ssl.h*):

```
int SSL_library_init(void);
```

Функция регистрирует все доступные шифрсьюты и возвращает 1 в случае успешного завершения.

Инициализация таблицы текстовых сообщений об ошибках осуществляется вызовом функции ***SSL_load_error_strings***, имеющей следующий прототип:

```
void SSL_load_error_strings(void);
```

Пример получения сообщения о произошедшей ошибке будет приведен ниже.

Также в начале программы можно инициализировать интерфейс WinSock API с помощью функции ***WSAStartup***, так как TLS-соединение будет создаваться на базе уже существующего сетевого соединения (сокета), поддерживающего передачу данных в соответствие с протоколом TCP.

Примечание. В данной работе не будут рассматриваться сигнатура и принцип действия используемых функций интерфейса WinSock API. Получить о них подробную информацию можно, например, в соответствующем разделе MSDN (https://msdn.microsoft.com/en-us/library/windows/desktop/ms741394%28v=vs.85%29.aspx).

Структура простейшего клиентского приложения, передающего и принимающего данные по протоколу TLS

После первоначальной инициализации клиентское приложение должно создать защищенное соединение с сервером и организовать передачу и прием данных. Для этого необходимо выполнить следующие действия:

1. Создать контекст TLS – переменную типа *SSL_CTX* (структура, определенная в файле *openssl/ssl.h*). Установить нужные параметры контекста.

2. Для созданного контекста задать параметры хранилища доверенных сертификатов центров сертификации, используемых при верификации сертификата сервера.
3. Создать объект TLS-соединения – переменную типа *SSL* (тип также определен в файле *openssl/ssl.h*).
4. С помощью функций интерфейса WinSock API (***socket***, ***bind***, ***connect***) открыть обычный сокет и установить логическое соединение по протоколу TCP с заданным сервером (к этому моменту сервер уже должен «слушать» указанный порт на предмет входящих клиентских соединений).
5. Установить полученный на предыдущем шаге дескриптор сокета в объект TLS-соединения, созданный в п. 3.
6. Инициировать установление TLS-соединения (запуск *TLS Handshake Protocol* – протокола «рукопожатия» или «хендшейка», целью которого является аутентификация сервера, обмен информацией о применяемых сторонами алгоритмах шифрования и хэширования, а также согласование ключевого материала).
7. В случае успешного завершения хендшейка, организовать передачу и прием необходимых данных. По окончании передачи разорвать TLS-соединение и закрыть сокет.
8. Освободить объект TLS-соединения и контекст TLS.

Далее будут рассмотрены функции библиотеки OpenSSL, необходимые для выполнения этих операций.

Функция ***SSL_CTX_new*** создает новый контекст TLS. Имеет следующий прототип:

```
SSL_CTX *SSL_CTX_new(const SSL_METHOD *meth);
```

Параметр *meth* указывает, какие версии протокола TLS (или SSL) будут поддерживаться. Существует ряд функций, которые возвращают константный указатель на структуру *SSL_METHOD*. В данной работе и в клиентском и в серверном приложении будем использовать функцию:

```
const SSL_METHOD *TLSv1_method(void);
```

Для созданного контекста можно устанавливать ряд параметров с помощью соответствующих функций и их опций. В нашем случае можно установить один параметр с помощью функции:

```
long SSL_CTX_set_mode(SSL_CTX *ctx, long mode);
```

При вызове функции зададим параметр *mode* равным константе *SSL_MODE_AUTO_RETRY*. В нашем случае клиентское приложение

для простоты будет устанавливать блокирующее соединение с сервером. Если этот параметр установлен, то при блокировании транспортного сокета, на базе которого устанавливается TLS-соединение, функции чтения и записи не будут возвращать ошибку и, прозрачно для приложения, будет проводиться повторная процедура хендшейка.

Ниже показан пример создания контекста TLS и установления данного параметра:

```
SSL_CTX *ssl_ctx = SSL_CTX_new(TLSv1_method());
SSL_CTX_set_mode(ssl_ctx, SSL_MODE_AUTO_RETRY);
```

Для проверки сертификата, переданного клиенту сервером, необходимо установить параметры хранилища доверенных сертификатов, которое может представлять собой папку или файл на диске. Путь к ним можно задать с помощью функции:

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx,
                                  const char *CAfile,
                                  const char *CApath);
```

Параметр *ctx* является указателем на созданный ранее контекст TLS. Параметр *CAfile* содержит имя файла, в котором находится сертификат корневого центра сертификации или последовательно перечисленный набор сертификатов в PEM-формате, входящих в доверительную цепочку (сюда же могут быть добавлены списки отозванных сертификатов *CRL*). В нашем случае для аутентификации сервера (верификации переданного им сертификата) будет использоваться единственный сертификат корневого центра сертификации (ранее сгенерированный и сохраненный в файле *ca_gost_cert.pem*). Строку с его именем необходимо передать в качестве параметра *CAfile*. В этом случае в качестве параметра *CApath* передается значение NULL. Функция возвращает 1 в случае успешного завершения.

Создание объекта TLS-соединения можно выполнить с помощью функции:

```
SSL *SSL_new(SSL_CTX *ctx);
```

Для заданного контекста TLS создается переменная типа *SSL*, указатель на которую возвращается функцией. В случае ошибки возвращается значение *NULL*.

После того, как с помощью функций интерфейса WinSock API будет открыт обычный сокет и установлено логическое соединение по протоколу TCP с сервером по заданным IP-адресу и номеру порта,

необходимо установить дескриптор сокета в объект TLS-соединения вызовом функции:

```
int SSL_set_fd(SSL *s, int fd);
```

Параметр s указывает на ранее созданный объект TLS-соединения, а *fd* содержит дескриптор открытого сокета, связанного с TCP-соединением. Функция возвращает 1 при успешном завершении.

После установки дескриптора сокета в объект TLS-соединения можно запускать процедуру хендшейка с помощью функции:

```
int SSL_connect(SSL *ssl);
```

Во время работы данной функции происходит, в том числе, верификация сертификата, переданного сервером, с помощью содержимого хранилища доверенных сертификатов, чьи параметры были установлены для контекста TLS ранее с помощью вызова функции ***SSL_CTX_load_verify_locations***. В случае удачного завершения хендшейка функция возвращает 1. Нулевое или отрицательное значение свидетельствует об ошибке. Ниже показан пример получения текстового описания ошибки в случае неудачи при установлении соединения с сервером. Код ошибки можно получить с помощью вызова функции ***SSL_get_error***.

```
int nRet;
if ((nRet=SSL_connect(ssl))<=0)
{
    char buffer[500];
    ERR_error_string(SSL_get_error(ssl, nRet), buffer);
    printf("Ошибка: %s\n", buffer);
}
```

После того, как соединение установлено, клиент может передавать данные серверу и получать их от него. И клиент и сервер используют для этого одни и те же функции. Для чтения данных из TLS-соединения используется функция:

```
int SSL_read(SSL *ssl, void *buf, int num);
```

Из указанного соединения (параметр *ssl*) считываются данные объемом не более *num* байт и помещаются в буфер *buf*. Функция возвращает количество реально прочитанных из соединения байт. Нулевое или отрицательное возвращаемое значение свидетельствуют об ошибке, код которой можно получить с помощью функции ***SSL_get_error***.

Для записи данных используется функция:

```
int SSL_write(SSL *ssl, const void *buf, int num);
```

Данные объемом *num* байт считываются из буфера *buf* и записываются в соединение *ssl*. Функция возвращает количество реально записанных байт. Диагностирование ошибки осуществляется также как и у функции ***SSL_read***.

Для того чтобы разорвать TLS-соединение используется функция:

```
int SSL_shutdown(SSL *s);
```

Функция отправляет соответствующее сообщение другой стороне о закрытии соединения. В случае успешной отправки сообщения и получения ответа от другой стороны функция вернет 1. Это будет свидетельствовать, что обе стороны выполнили данную функцию. Если ответ не получен, то вернется значение нуля. В этом случае можно повторно выполнять функцию до получения 1, что будет означать закрытие соединения другой стороной. Функция вернет значение -1 в случае ошибки, код которой можно будет извлечь с помощью вызова ***SSL_get_error***.

После разрыва TLS-соединения приложение может закрывать TCP-соединение и сокет с помощью функций интерфейса WinSock API ***shutdown*** и ***closesocket*** соответственно. Далее освобождается объект TLS-соединения с помощью вызова функции:

```
void SSL_free(SSL *ssl);
```

Данная функция уменьшает на единицу количество ссылок на указанный объект и уничтожает его после обнуления этого количества. Освободить контекст TLS можно вызовом функции:

```
void SSL_CTX_free(SSL_CTX *);
```

Принцип работы данной функции аналогичен функции ***SSL_free***.

Структура простейшего серверного приложения, принимающего и передающего данные по протоколу TLS

Для работы простейшего серверного приложения необходимо реализовать следующий алгоритм:

1. Создать контекст TLS и установить его параметры аналогично клиентскому приложению.
2. Установить в контекст сертификат сервера и его закрытый ключ.
3. С помощью функций интерфейса WinSock API (***socket***, ***bind***) открыть обычный сокет для установления соединения по

протоколу TCP. Перевести созданный сокет в режим прослушивания клиентских соединений вызовом функции ***listen***.

4. После успешного завершения функции ***listen***, вызвать функцию ***accept***, которая ожидает появление входящего соединения и возвращает новый подключенный сокет. Создать новый объект TLS-соединения и установить в него (аналогично клиентскому приложению) возвращенный функцией ***accept*** дескриптор сокета.
5. Перевести TLS-соединение в режим ожидания инициирования хендшейка клиентом. После его успешного завершения организовать прием и передачу необходимых данных. По окончании разорвать TLS-соединение и закрыть сокет.
6. Освободить объект TLS-соединения и контекст TLS.

Для того чтобы установить сертификат сервера в контекст TLS можно использовать функцию:

```
int SSL_CTX_use_certificate_file(SSL_CTX *ctx,
                                 const char *file,
                                 int type);
```

В качестве параметра *file* нужно передать строку с именем файла, содержащего сертификат сервера (в нашем случае *serv_cert.pem*). Для того чтобы показать, что файл содержит сертификат в формате PEM, параметр *type* нужно задать равным константе *SSL_FILETYPE_PEM*. Функция возвращает значение 1 в случае удачного завершения и ноль в случае ошибки.

Установить в контекст сертификата закрытый ключ можно с помощью функции:

```
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx,
                                const char *file,
                                int type);
```

Назначение параметров данной функции аналогично таковым у рассмотренной выше функции ***SSL_CTX_use_certificate_file***. При загрузке закрытого ключа (в нашем случае файл *serv_key.pem*) функция потребует ввести в консоли пароль. Если же пароль необходимо задавать в тексте программы, то перед вызовом данной функции можно выполнить функцию:

```
void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx,
                                            void *u);
```

Эта функция задает пароль, который будет использоваться при доступе к закрытому ключу функцией ***SSL_CTX_use_PrivateKey_file***. В качестве параметра *u* передается строка с паролем максимальной длины 255 символов.

После установления с клиентом TCP-соединения и создания объекта TLS-соединения, его необходимо перевести в режим ожидания запроса клиента на проведение хендшейка. Это делается с помощью функции:

```
int SSL_accept(SSL *ssl);
```

Функция вернет значение 1 при успешном завершении и нулевое или отрицательное значение в случае ошибки (пример того, как получить информацию об ошибке был рассмотрен при описании функции ***SSL_connect***).

После успешного установления соединения сервер может принимать и передавать информацию с помощью ранее описанных функций ***SSL_read*** и ***SSL_write***. По окончании обмена данными производятся те же действия по разрыву TLS и TCP соединений, закрытию сокета, освобождению объекта TLS-соединения и контекста TLS, что и в клиентском приложении.

СОДЕРЖАНИЕ РАБОТЫ

1. С помощью криптографического пакета OpenSSL создать:
   - ключевую пару алгоритма ГОСТ Р 34.10-2001 с набором параметров *"A"* (RFC 4357) и соответствующий ей самоподписанный сертификат центра сертификации;
   - ключевую пару алгоритма ГОСТ Р 34.10-2001 с набором параметров *"A"* и соответствующий ей сертификат в PEM-формате для сервера. Сертификат должен быть подписан закрытым ключом центра сертификации.
2. Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольные клиентское и серверное приложения (или одно приложение, имеющее два режима), предназначенные для создания блокирующего соединения по протоколу TLS и выполняющие обмен сообщениями. Клиентское приложение должно запрашивать у пользователя IP-адрес в десятично-точечной форме. Номер порта (произвольное значение больше 1023) определяется в приложениях заранее. В качестве очередного сообщения должна выступать символьная строка заранее оговоренной максимальной длины, которую

пользователь клиентского приложения вводит в консоли, после чего она передается серверу. Сервер, получив строку, выводит ее в консоль и возвращает ее клиенту. Клиент также для подтверждения приема выводит строку в консоль. Строка должна передаваться вместе с ограничивающим ее длину нуль-символом (0). Передача клиентом строки нулевой длины (то есть состоящей только из одного нуль-символа) будет означать окончание сеанса связи.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как в протоколе TLS осуществляется аутентификация сервера и клиента?
2. Как с помощью криптографического пакета OpenSSL осуществить генерацию ключевой пары алгоритма ГОСТ 34.10-2001 и создание самоподписанного сертификата?
3. Что собой представляет обобщенный алгоритм работы клиентского приложения, передающего и принимающего данные по протоколу TLS?
4. Какие функции WinSock API используются для открытия и закрытия сокетов, создания и разрыва TCP-соединений?
5. Как для клиентского приложения установить параметры хранилища доверенных сертификатов?
6. Что такое TLS Handshake Protocol?
7. Какие функции OpenSSL используются для создания и установки параметров контекста TLS?
8. Как создать объект TLS-соединения и связать его с сокетом, поддерживающим TCP-соединение?
9. Как разорвать TLS-соединение и освободить его объект и контекст TLS?
10. Как клиентское приложение может инициировать процедуру хендшейка?
11. Какие функции OpenSSL используются для передачи и приема данных по протоколу TLS?
12. Что собой представляет обобщенный алгоритм работы серверного приложения, передающего и принимающего данные по протоколу TLS?
13. Как установить в контекст TLS серверного приложения сертификат сервера и его закрытый ключ?
14. Как перевести серверное приложение в режим ожидания запроса клиента на проведение хендшейка?

ЗАКЛЮЧЕНИЕ

Развитие информационных технологий привело к тому, что стали стремительно расти ценность и значимость информационных ресурсов. Основным требованием к современным информационным системам стало обеспечение доступности, целостности и конфиденциальности информационных ресурсов. В реализации таких требований большую роль играет комплексный подход, сочетающий в себе меры законодательного, организационного и программно-технического характера. К числу последних относятся средства криптографической защиты информации, которые получают все большее распространение при построении информационных систем.

В пособии были рассмотрены различные подходы к использованию криптографических алгоритмов. Студентам предложено осуществить самостоятельную реализацию потокового шифра на базе регистра сдвига с линейной обратной связью и блочного шифра ГОСТ 28147-89. Также учтен тот факт, что в современных условиях разработчик программного обеспечения, как правило, использует готовые криптографические интерфейсы и пакеты, содержащие в себе достаточно обширный функционал. Поэтому в пособии были рассмотрены особенности применения криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation для симметричного шифрования и создания криптографических сообщений.

Также студентам предлагалось получить навыки в использовании одного из наиболее популярных криптографических пакетов с открытым исходным кодом OpenSSL. В трех лабораторных работах рассматривались вопросы симметричного и асимметричного шифрования, создания цифровых сертификатов X.509, а также реализации сетевых приложений с поддержкой защищенного протокола передачи данных TLS.

Полученные знания и навыки позволят студентам в дальнейшем успешно применять средства криптографической защиты информации в составе разрабатываемых информационных систем и другого программного обеспечения.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Бернет, С.* Криптография. Официальное руководство RSA Security / С. Бернет, С. Пэйн. – М.: Бином-Пресс, 2002. – 384 с.

2. *Лапонина, О.Р.* Основы сетевой безопасности. Криптографические алгоритмы и протоколы взаимодействия / О.Р. Лапонина. – М.: Бином. Лаборатория знаний, 2009. – 536 с.

3. *Масленников, М.Е.* Практическая криптография / М.Е. Масленников. – СПб.: БХВ-Петербург, 2003. – 458 с.

4. *Молдовян, А.А.* Криптография / А.А. Молдовян. – СПб.: Лань, 2001. – 224 с.

5. *Панасенко, С.П.* Алгоритмы шифрования. Специальный справочник / С.П. Панасенко. – СПб.: БХВ-Петербург, 2009. – 576 с.

6. *Рябко, Б.Я.* Основы современной криптографии для специалистов в информационных технологиях / Б.Я. Рябко. – М.: Научный мир, 2004. – 172 с.

7. *Смарт, Н.* Криптография / Н. Смарт. – М.: Техносфера, 2005. – 528 с.

8. *Столлингс, В.* Криптография и защита сетей: принципы и практика. – 2-е изд. / В. Столлингс. – М.: Вильямс, 2001. – 672 с.

9. *Фергюсон, Н.* Практическая криптография / Н. Фергюсон. – М.: Диалектика, 2005. – 432 с.

10. *Шаньгин, В.Ф.* Защита информации в компьютерных системах и сетях / В.Ф. Шаньгин. – М.: ДМК Пресс, 2012. – 592 с.

11. *Шнайер, Б.* Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си / Б. Шнайер. – М.: ТРИУМФ, 2003. – 815 с.

12. *Щербаков, А.Ю.* Прикладная криптография. Использование и синтез криптографических интерфейсов / А.Ю. Щербаков, А.В. Домашев. – М.: Русская редакция, 2003. – 416 с.

13. *Щупак, Ю.А.* Win32 API. Эффективная разработка приложений / Ю.А. Щупак. – СПб.: Питер, 2007. – 572 с.

14. *Ященко, В.В.* Введение в криптографию / В.В. Ященко. – М.: МЦНМО, 2012. – 352 с.

15. Средство криптографической защиты информации. МагПро КриптоПакет вер. 2.1. Библиотека libcrypto. Руководство программиста [Электронный ресурс]. – 2012. – Режим доступа: http://www.cryptocom.ru/docs/cryptopack21-libcrypto.pdf

16. Средство криптографической защиты информации. МагПро КриптоПакет вер. 2.1. Библиотека libssl. Руководство программиста [Электронный ресурс]. – 2012. – Режим доступа: http://www.cryptocom.ru/docs/cryptopack21-libssl.pdf

17. Средство криптографической защиты информации. МагПро КриптоПакет вер. 2.1. Утилита OpenSSL. Руководство оператора [Электронный ресурс]. – 2012. – Режим доступа: http://www.cryptocom.ru/docs/cryptopack21-openssl.pdf

18. Технологии и продукты Microsoft в обеспечении информационной безопасности. Национальный Открытый Университет «ИНТУИТ». [Электронный ресурс]. – 2010. – Режим доступа: http://www.intuit.ru/studies/courses/600/456/info

Учебное издание

Смышляев Артем Геннадьевич
Жуков Алексей Владимирович
Жуков Иван Владимирович

МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Лабораторный практикум
для студентов очной формы обучения направления
бакалавриата 09.03.02 – Информационные системы и технологии

В авторской редакции